University of Leicester

CMMI-CM COMPLIANCE CHECKING OF FORMAL BPMN MODELS USING MAUDE

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Nissreen A. S. El-Saber, MSc. (Cairo University)

Department of Computer Science

University of Leicester

December 2014

CMMI-CM COMPLIANCE CHECKING OF FORMAL BPMN MODELS USING MAUDE

Nissreen El-Saber

ABSTRACT

From the perspective of business process improvement models, a business process which is compliant with best practices and standards (e.g. CMMI) is necessary for defining almost all types of contracts and government collaborations. In this thesis, we propose a formal pre-appraisal approach for Capability Maturity Model Integration (CMMI) compliance checking based on a Maude-based formalization of business processes in Business Process Model and Notation (BPMN). The approach can be used to assess the designed business process compliance with CMMI requirements as a step leading to a full appraisal application. In particular, The BPMN model is mapped into Maude, and the CMMI compliance requirements are mapped into Linear Temporal Logic (LTL) then the Maude representation of the model is model checked against the LTL properties using the Maude's LTL model checker.

On the process model side, BPMN models may include structural issues that hinder their design. In this thesis, we propose a formal characterization and semantics specification of well-formed BPMN processes using the formalization of rewriting logic (Maude) with a focus on data-based decision gateways and data objects semantics. Our formal specification adheres to the BPMN standards and enables model checking using Maude's LTL model checker. The proposed semantics is formally proved to be sound based on the classical workflow model soundness definition. On the compliance requirements side, CMMI configuration management process is used as a source of compliance requirements which then are mapped through compliance patterns into LTL properties. Model checking results of Maude based implementation are explained based on a compliance grading scheme. Examples of CMMI configuration management processes are used to illustrate the approach.

Acknowledgements

In the name of Allah, The Almighty, the Most Gracious, the Most Merciful, I thank Allah, who always bless me and provided me with great people to work with. I thank them all who made this thesis possible. To my supervisor, Dr. Artur Boronat, who I have learnt a lot from and without his support I would not be able to write this thesis. To Prof. Reiko Heckel, I used to leave his office after our biannual meetings full of ideas and possible solutions for my research obstacles. Many thanks to the teaching and administrative staff of the University of the Leicester, UK for providing a comfortable working environment. In particular, I am thankful to Dr. Fer-Jan De Vries for his valuable discussions, Mr. Gavin Hornsey and Ms Karen Smith for their support and cooperation during my PhD. Many thanks to all my friends who were always there to help me whenever I needed them.

My loving husband, Mohamed Mansoura, who kept encouraging me and believing in me all the way. Thank you! My Zahia, the blessing Allah provided me with during my PhD. Although the PhD took me from her most of the time, Zahia fuels me up with determination, confidence and hope in our little time together. My warmest gratitude to Mahjabin Qadri, Aunty Siraad, Rahma nursery, CAPTA nursery who took care of Zahia for over three years .. Jazakom Allah-u khayran!

To my mother, be happy and proud in the Heaven, Amen. To my father, who called me Professor since I joined the university as an undergraduate student. Well, I am not a professor yet, but I am doing my best. I promise to do it one day! Nermen, nothing can describe a sister like you, thank you, and it is your turn now! Solieman, my soul mate, I enjoyed our little helpful technical discussions. Nasser, you always made me smile when we talk. The hope you fill me with is incomparable. To my big family in Egypt thank you for your prayers, encouragement and support.

Nissreen El-Saber

Leicester, March 2014

Contents

List of Tables

List of Figures

Abbreviations

BP	**B**usiness **P**rocess
BPMN	**B**usiness **P**rocess **M**odel and **N**otation
CM	**C**onfiguration **M**anagement
CMMI	**C**apability **M**aturity **M**odel **I**ntegration
CMMI-CM	**CMMI** - **C**onfiguration **M**anagement process area
LTL	**L**inear **T**emporal **L**ogic
OMG	**O**bject **M**anagement **G**roup

Dedicated to Egypt . . .

Declaration

The content of this submission was undertaken in the Department of Computer Science, University of Leicester, and supervised by Dr. Artur Boronat and Prof. Reiko Heckel during the period of registration. I hereby declare that the materials of this submission have not previously been published for a degree or diploma at any other university or institute. All the materials submitted for assessment are from my own research, except the reference work in any format by other authors, which are properly acknowledged in the content. Part of the research work presented in the following:

1. Nissreen El-Saber and Artur Boronat, *BPMN Formalization and Verification using Maude*. The 6th Workshop on Behavioural Modelling - Foundations and Applications (BM-FA 2014). York, United Kingdom, 22 July 2014. In collaboration with ECMFA 2014.

   This paper presents the formal syntax and semantics for BPMN models using Maude. The resulting models are proved to be sound based on the classical soundness for workflow models. The work included in the paper is in Chapter 3 and Chapter 4.

2. Nissreen El-Saber and Artur Boronat, *A Maude based Formalization for BPMN Models*. Post-proceedings of six International Workshops on Behaviour Modelling-Foundations and Applications, LNCS, July 2015. (in progress)

   This paper extends the BMFA 2014 paper with formalizing more elements of the BPMN. The formal syntax and semantics is used to model configuration management processes using Maude. The paper includes parts of Chapter 3, Chapter 4, and Chapter 5.

Chapter 1

Introduction

This chapter presents the motivation for the research presented in this thesis in Section 1.1, followed by the thesis main statement and hypotheses in Section 1.2. The proposed approach for compliance checking of BPMN models is introduced in Section 1.3. After that we justify each language, tool, and technique used to demonstrate the approach in Section 1.4, followed by a list of contributions in Section 1.5. Finally, Section 1.6 outlines the remaining chapters.

1.1 Overview and Motivation

Compliance is defined in [32] as ascertaining and proving the adherence of business processes to relevant accepted standards, code of practices, legislations and laws, internal policies and business partner contracts. Following standards and regulations allows businesses to have more disciplined and monitored activities. There are many sources for the standards and regulations; some are external and others are internal within the organization [34, 32]. External compliance requirements can come from laws and legislations within the country or worldwide if trading globally, the information security standards (e.g. ISO 27001 [72]), or process improvement specifications models (e.g. CMMI [22], IEEE [85]) while all the rules and procedures that are customized by the company and are used internally for efficiency are considered internal sources.

Business process improvement models provide an identification and understanding of the designed process and its implementation in order to ensure that it is aligned with customer needs/expectations as well as the quality measures. A business process (BP) which adheres to some related improvement models is known to be more effective, efficient, and transparent [32, 6, 80]. In software development market, no matter how big or small the organization is, it follows one or more process improvement models to improve the quality of software (and/or services) developed.

For example, Capability Maturity Model Integration (CMMI) [22] is a collection of best practices developed by Carnegie Mellon University - Software Engineering Institute (CMU-SEI) with members from industry, government in order to help software organizations improve their processes. CMMI has a specific set of compliance checking methods (i.e. SCAMPI [83]) which proved to be expensive in terms of costs to Small and Medium-size Enterprises (SME) [66]. Configuration management (CM) process area is one business process aspect which the CMMI provides guidelines for. CM is concerned with establishing, documenting and monitoring the changes of the basic work items (i.e. products and/or services) within an organization. These activities are assessments which check the existence of documents and evidences of following certain standards as will be explained later. This includes using observations, interviews and questionnaires which normally consume huge amounts of the working hours without production. In order to increase the chances of successful appraisals and to reduce the time and costs of un-

derstanding the organization process to deliver its products/services and to adhere to its contracts, we propose an algebraic automatic approach to formally check the compliance of the organization BP design with the set of best practices described by the CMMI model in Configuration Management (CM) Process Area.

Despite the increasing number of compliance checking methods and tools (e.g. [64, 80, 49, 32, 6]), organizations are still facing difficulties in finding effective support to ensure that their BPs comply with the improvement requirements. In many cases, manual solutions are being used to assure quality in BP and these are consuming time and costs and offer limited assurance for compliance [80, 42]. Partially or fully automating manual BP inspections and audits would substantially reduce the overall cost of compliance [42]. For the automation to take place, the BP and the compliance requirements should be formally represented and a formal assessment procedure to check if a BP satisfies the formalized requirements. Despite being formal, the procedure should be accessible to business people.

In Table 1.1, a comparison among the state-of-the-art approaches in checking business process compliance with standards and regulations is summarized. It compares them with respect to the formal languages used for system specification, property specification, and checking procedure. The classification column is assigning each approach with its type with respect to forward/backward compliance checking classification in [80] and explained in Chapter 6. The table provides information about the application domain and automation means of each approach. As seen from the table, the work regarding CMMI compliance checking is limited (e.g. [24]) and does not include any formal representation or tool automation. More details about the approaches are provided in Chapter 6.

Generally, BPs can be modelled using different notations, in either formal or informal representations. From a computer science point of view, formal modelling languages are more reliable and verifiable, while from a business point of view, non-technical users usually prefer to use informal graphical modelling languages. Not all formal modelling languages have accessible graphical representations (e.g. CSP [104],π-calculus [75]), while not all graphical languages have a comprehensive formalization (e.g. BPMN [68]). The accessible graphical notations for modelling BPs need to have the back-end for-

Table 1.1: Summary comparison of some compliance checking approaches

Ref.	**System Spec**	**Property Spec**	**Check Proc.**	**Class.**	**App. Domain**	**Automation**
[6]	BPMN 2 PetriNets	BPMN-Q 2 PLTL	MC	FD	Banking	Oryx, Lola
[33]	BPEL	FCL 2 LTL	MC	F,B	Sarbens-Oxley Act	COMPAS
[94]	MXML	LTL	MC	B	event-logs	ProM
[24]	**MDD**	**BPRE4OO**	**SCAMPI**	**FDT**	**CMMI-DEV**	**NA**
[38]	EPC 2 PetriNets	ITIL adopted models	MC	FD	ITIL	ProM
[106]	Z	LTL	MC	F	ISO/IEC 15408	FORVEST
[49]	BPEL 2 π-calculus	BPSL 2 LTL	MC	FR	ITIL,COBIT	OPAL

Legend: D: Design-time, R: Run-time, MC: Model Checking, F: Forward, B: Backward, NA: Not Available, PM: Process Mining

malization that allows for the validation and verification of the designed models [10]. Focusing on the BPMN as a modelling language, one can find a number of issues with its standards and available formalizations (c.f. Section 1.4). In this work, we propose a rewriting logic-based formalization in Maude for BPMN models.

On formally representing the compliance requirements, we agree with the authors in [64, 32, 51, 42] on the need to separate the formalization of the designed BP from the formalization of the requirements. Decoupling these two aspects allows for more clear, unbiased assessment of the BPs without possible enforcing them to be compliant as it is difficult for many companies to change their business process in short time for the reasons of an assessment. Moreover, most SME does not really need to apply all the requirements as they are advisory, however, by breaking down the requirements into smaller requirements and checking their relevance and satisfaction one by one can allow the modeller to exclude the irrelevant requirements (e.g. a document which is not used in the process being checked). The standards are usually written in natural language which makes the translation of the standards into formal expressions a time and effort consuming process. Therefore, trying to add formality to the standard requirements, we use an intermediate step, i.e. mapping into compliance patterns [33], to formally represent the textual requirements in CMMI with Linear Temporal Logic (LTL).

1.2 Research Hypotheses

The main research statement*: Is it possible to formally check the BPMN models compliance against the CMMI Configuration Management requirements?*

In order to provide an answer to the main research statement above, we need first to identify the elements that we are going to deal with. Extracting from the research statement, there should be a formal BPMN model for a SME, a formal representation of the CMMI-CM requirements, and a formal compliance checking technique. Based on the following hypotheses, we introduce a compliance checking approach for BPMN models with CMMI-CM requirements.

1. ***What formalization of BPMN models can be considered suitable for compliance checking? If there is no such formalization, what are the main characteristics of a candidate formalization? Which formal language to use?***

   BPMN is a well-known modelling notation for BPs which is accessible to business users as well as computer scientists. It has been formalized using different formal languages (e.g. [27, 107, 104, 74, 28, 40]), however, most of these formalizations suffer from unclear semantics [68], lack of data object representation [27, 107, 104, 74, 28, 40], and non-determinism in decision-based gateways semantics [107]. Therefore, a new formalization is required which should cover the following: (1) consider core BPMN elements [68] (e.g. activities, gateways, events, data objects), (2) a comprehensive semantics for data objects as process resources, (3) a formal representation and evaluation of decision-based gateways guard expressions, and (4) a sound semantics for the BPMN models. Maude is an expressive declarative logical language for concurrent processes based on rewriting logic which can formally represent BPMN models. Moreover, it has its own verification toolkit which includes variety of tools, e.g. LTL model checker.

2. ***What are the characteristics of the CMMI process improvement model that make it an interesting area for compliance checking? Is it possible to formally represent the CMMI requirements? How?***

   Process improvement models are generally described in natural languages making the measuring and assessment of their applications a subjective process [32, 6, 42]. Moreover, they are lengthy and complex which decreases understandability and increases difficulties of manual compliance checking [64], e.g. CMMI [21], ISO [45], and IEEE [85]. CMMI certified SMEs are believed to have stable, continually improved BP as well as gaining more worldwide contracts with other organizations [22, 24, 86, 37]. In particular, for its importance, special focus on software SMEs and due to the lack of comprehensive formal compliance checking for it, we are focusing on CMMI [21]. Nevertheless, formally representing the CMMI requirements requires a property specification language. Considering their expressiveness and intuitive appeal and recommendations in related research [6, 32, 96], temporal logic can be used as the CMMI requirement specification language.

3. ***What is the verification technique to formally check compliance of BP models with formal requirements? Is it able to provide an explicit answer to the question: "Is an input process compliant with the input set of properties?"?***
   Model checking is popular for debugging and verification purposes [9, 46]. The structure of the problem is more like a model checking problem where the BP is the system model M, the requirements are properties Φ, and the checking is modelled as $M \models \Phi$ which is a model checking procedure. Model checking technique possibly can be unable to provide such decision. It gives a counterexample (i.e. a possible trace where the property is not satisfied) but does not give details on the nature of the property itself. Therefore, the compliance checking approach should provide informal suggestions for the modeller to modify their BP model.

4. ***What are the automation possibilities of the compliance checking approach?***
   Many challenges are facing the compliance checking automation in general and in particular designing this approach. For example, the CMMI requirements may have more than one formal representation. Furthermore, interpreting the model checking results (i.e. $true$ or $counterexample$) in the context of compliance checking requires a mapping grading scheme which represent a single requirement weight with respect to other requirements. Although the possibility for fully automating the compliance checking process is limited, there is still a promising opportunity to automate parts of the compliance checking process to benefit from formal specifications and automatic verification. We provide a Maude based tool automating most of the approach steps. An overview of the automation of the approach is discussed in Chapter 7.

1.3 Proposed Approach

We aim at giving an organization an idea about how mature its business process model is with respect to the CMMI reference model through applying a *pre-appraisal* method prior to going through the formal CMMI appraisals (e.g. SCAMPI). Our proposed approach is a formal automatic compliance checking method for the CM process area. This approach can be used as a *pre-appraisal* method to check how ready the designed

business process is to go for the expensive appraisal methods. The approach uses the company's designed BP process, transform it into Maude following the syntax and semantics presented in Chapter 3 and check its compliance with the CMMI-CM practices. Based on this check results, a designed process can be deemed to be CMMI-CM compliant or not, indicating which practices need to be improved, with initial information about which properties are not satisfied. This is illustrated in Figure 1.1.

Assuming that the SME is following their designed CM process (or the designed process reflects what is actually being done in the SME), the proposed approach allows the SME to start building their stable CM process which can be ready for more advanced form of appraisal (i.e. SCAMPI). In Figure 1.1, the approach consists of three basic parts: (1) BPMN2MAUDE procedure which provides the system specifications and is presented in Chapter 3, (2) CMMI-CM2LTL procedure which provides the property specifications in LTL for CMMI-CM process area and is presented in Chapter 5, and (3) Model Checking procedure which we customised to fit into our compliance checking domain of application in Chapter 5.

The compliance checking is considered a model checking problem, hence, three components should be elaborated; the system specifications, the property specifications and the model checking procedure. On one side, the system specification part (i.e. BPMN2MAUDE) starts with a BP model in BPMN which is then mapped into the proposed BPMN syntax and semantics in Maude. The BP model in Maude is then checked if well-formed based on the well-formedness property definition in Definition 3.3.2. The well-formed BPMN models are proves to be sound in Chapter 4. If the process model is not well-formed, then the modeller will have information about which set of objects in the model is not well-formed to update the model and run the well-formedness test again. This part of the approach is explained in Chapter 3. In case the BP model is well-formed, then it is ready for model checking.

On the other side lies the CMMI, which requirements (i.e. sub-practices) are the basic component for the property specification (i.e. CMMI-CM2LTL). As will be explained in Chapter 2, CMMI staged representation maturity level (ML) consists of process areas (PA), and each process area contains specific goals (SG) and each specific goal contains specific practices (SP), and each specific practice has a number of sub-practices [21]

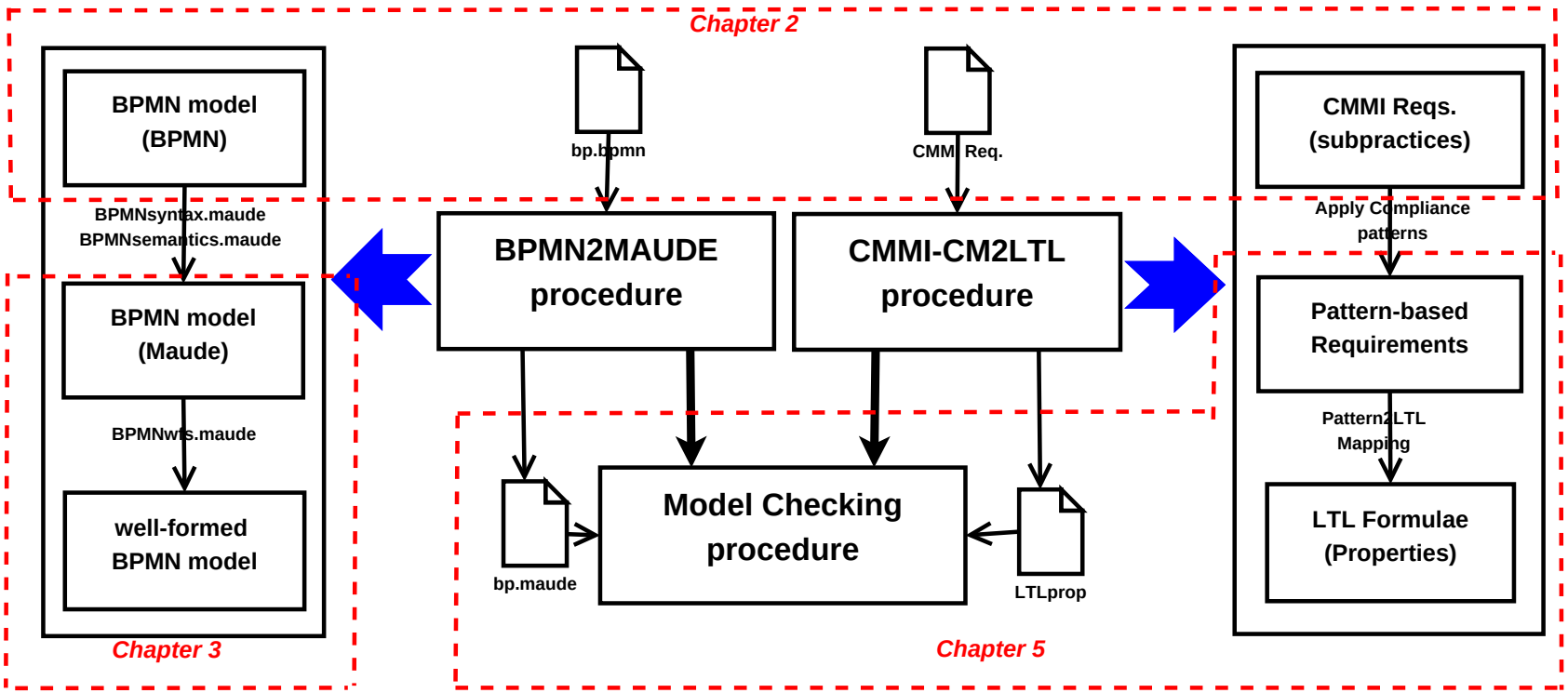


Figure 1.1: The Proposed Compliance Checking Approach

which are the textual requirements of the CMMI. These sub-practices are extracted and represented using the compliance patterns [33, 64]. As the patterns has a direct mapping into LTL formulae (as in [33]), the pattern-based requirements are represented as LTL formulae, ready for the model checking. This part of the approach is explained in Chapter 5.

The sub-practices under a specific practice are modelled to a set of LTL properties $\Phi_{PA} = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ for n the number of modelled properties while the mapping relation is not necessarily a one-to-one relationship. That is, the number of sub-practices in a certain specific practice does not necessarily equal to the number of properties formulae modelled. A specific goal is satisfied if its underlying specific practices are satisfied. Therefore, for a system model M representing the BP and the set of properties representing the specific goal, the model checking problem (i.e. satisfaction relation) becomes $M \models \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$. Generally, the set of modelled properties for a process area (e.g. CM) is represented as Φ_{CM}. A model M satisfies the process area CM iff all properties $\varphi_i \in \Phi_{CM}$ are satisfied in M, i.e. $\forall \varphi \in \Phi_{CM}\ (M \models \varphi) \Rightarrow M \models \Phi_{CM}$.

However, reference models such as process improvement models are advisory and some companies may try to focus more on some requirements than others, or are applying a subset of the requirements. That is, if checking the properties with the and operator (i.e. $\wedge$), then the result is going to be *false* if at least one property is not satisfied, although the process might be compliant. Therefore, we introduce a compliance grading scheme which allows for quantifying the model checking results. Moreover, customization function is used to allow model checking for properties which are not spurious as shown in Section 5.3.2.

Finally, the well-formed BPMN model in Maude is fed into Maude LTL model checker [30] as the system specification (i.e. the model M), and the LTL formal properties representing the CMMI requirements as the properties to be checked against the model. The model checking results are analysed for spurious (i.e. fake) outputs as discussed in Chapter 5.

1.4 A Brief Justification of Used Tools and Techniques

In the compliance checking approach, certain tools, languages, and techniques are used. In this section, we briefly describe the reasons which made them suitable to demonstrate our contributions. We justify the use of BPMN, Maude, CMMI, and model checking technique.

1.4.1 BPMN

The Business Process Modelling Notation (BPMN) is a widely used standard notation and model for representing business processes (BPs) in the design phase of systems development. It has been the basis for many BPs formalizations (e.g. [27, 107, 104, 74, 75, 28, 40]) which we believe are not fully representing its powerful elements; such as data objects and some control flow gateways (e.g. inclusive decision-based OR gateways) as explained later in Chapter 6. According to the OMG [68], 72 implementations of the BPMN are reported for known businesses (for example, Oracle). Some issues related to BPMN formalizations allow for ambiguous and unstructured BP models [10, 27, 40], such as:

1. The unclear semantics of different BPMN elements allow for incompatibility in the design interpretations, analysis and use of BP models [10],

2. In spit of their importance as process resources, data objects formalization and usage are still under-represented (e.g. [27, 40, 105, 107]),

3. Although the non-determinism attached with the decision gateways representation helps to consider all possible alternative flows (e.g. in [27, 102, 90]), the ability to evaluate a guard conditional expression is needed to simulate a more reliable and effective gateway formal comprehensive representation.

These issues can be handled and/or avoided through a comprehensive formal syntax and semantics specifications for BPMN models. That is, some formal restrictions can be easily imposed on to the BPMN models that can reduce/avoid the structural and behavioural problems (e.g. misinterpretations, document lack of representation, deadlock and lack of synchronization).

1.4.2 Maude

Maude is a logical declarative language based on rewriting logic. It is designed for representing and reasoning about concurrent systems. The following reasons made the tandem formed by rewriting logic and its Maude implementation a very convenient setting for formalizing BPMN models.

1. Equational pattern matching, for example, if the left hand side pattern matched, then the right hand side is applied when the conditional being satisfied in case of conditional equations, memberships and rules.

2. User-definable syntax and data, i.e. Maude allows the user to define their own data types as well as the structural relations among the components in the system.

3. Types, subtypes, and partiality, i.e. the sorting subsorting relations allows for hierarchy of data types which provides the user with flexible use of operators and memberships.

4. Support for objects, i.e. Maude system modules supports the use of object oriented programming style with predefined class definitions. Moreover, the data types of information into these objects are user defined and can be generic.

5. Maude has its verification toolkit [20, 30] which can be used to verify models designed in Maude. The `Search` command is used to locate certain states in the model while the Maude LTL model checker [30] is used to verify properties of unwanted behaviours in the model.

6. Reflection, meta-representations and computations, i.e. Maude has many meta-language applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation.

7. Concurrent processes applications of Maude, i.e. there are already a number of successful Maude implementations for processes (e.g. [98, 26, 43, 61, 13]) which are promising and encouraging.

1.4.3 CMMI-CM

Many SME are part of larger organizations which are actively using CMMI to support their business goals. That is, the need for operational effectiveness and efficiency increases as the size of the organization grows. The same applies to projects within organization. Moreover, expectations for CMMI-compliance practices increases when partnering or subcontracting with larger companies. Even when independently bidding on some government business, CMMI-based appraisals are required [37]. CMMI can be a way for the small organizations to gain better projects and customers, by demonstrating their competency in process management while preserving their flexibility. Emphasising on the role of management in all its maturity levels, CMMI improves management visibility in software development [86]. This is achieved by improving controlling and monitoring different changes and audit the documentation process through reusing the already existed process, after improving it, to develop organization's software products/services a higher level of quality. Our approach aims at encouraging the SMEs to test their BPs adherence with process improvement models (i.e. CMMI) in a formal, yet less expensive in terms of cost, pre-appraisal compliance checking.

From the CMMI process areas, we focus on the configuration management (CM) process area for several reasons. First, CM is a support process area shared by the three areas of interest of the CMMI, i.e. development, services, and acquisition, which makes it a common requirement for any SME trying for CMMI appraisals. Second, CM is a process area belongs to the second maturity level (ML2) in the staged representation of the CMMI, i.e. the first level of formal accreditation for a SME to look for (more details about CMMI maturity levels can be found in Chapter 2). Third reason comes from the importance of a well-established CM process in the organization that seeks any kind of improvement as a basis for documenting, auditing and controlling changes in what could be called SME process infrastructure (e.g. configuration items, change requests, audits). Fourth, CM is a basic process area in most process improvement models (e.g. IEEE [85], ISO [45]) that are used by software companies. Although the basic requirements are identical, the detailed requirements may vary. We are focusing here on CMMI-CM.

1.4.4 Model Checking Technique

Besides the general strengths of model checking techniques (e.g. [9, 19]), there are more specific motivations concerning using the model checking for compliance checking problems. We include details about model checking technique in Chapter 5.

1. Model checking is a fully automated procedure [46, 9] which known to be fast [9] compared to conventional testing and simulations. If the inputs are ready, then the verification procedure works automatically, if initiated. For checking the compliance of process improvement model (i.e. CMMI), an automated formal procedure would be suitable to reduce different kinds of costs and benefiting from the mathematical and temporal basis for the procedure.

2. Model checking supports partial verification [46, 9]. That is, the properties can be checked one-by-one allowing for spotting the unwanted behaviour in details. Looking at the compliance checking problem, the modeller would like to get results indicating which properties are satisfied by the model and which are not. It makes the model checking a perfect candidate while it is not the case with other verification techniques (for example, Theorem proving).

3. Model checking does not require proofs to verify the properties [46, 9], which make it a suitable choice when the people involved in the compliance checking normally are not aware of proof methods or other formal verification techniques. Model checking provide the balance in this case.

4. Model checking produces diagnostic counterexamples in case a property is not satisfied with respect to the model specifications [46, 9]. It represents the trace of state transitions which can be used for debugging. This feature helps the modeller to investigate a possible trace of the model where the property does not hold.

5. The property specifications in model checking is mainly represented in Temporal logic, adding the powerful expressiveness of it in representing properties over traces generated by transition systems [19, 96]. Besides, compliance patterns have been developed to facilitate the mapping between the textual compliance requirements and the LTL formulae (e.g. [33]).

6. BP compliance checking is generally dealt with as a model checking problem (e.g. [64, 80, 49, 32, 6]), where the BP model is the system specifications being checked and the compliance requirements are the properties checked against that system.

7. The formalization language, Maude, has its own LTL model checker [30]. It combines an expressive and general system specification language (Maude [20]) with an LTL model checking engine. Maude uses the recent advances in on-the-fly explicit-state model checking techniques [31].

8. The model checking applications are more control-intensive and less data-intensive in nature due to the infinite range of data. Although the BPs can wait for a decision or a certain confirmation before proceeding in execution, we do not expect the system models to simulate the data contained in the expected artefacts. For example, the model may contain data objects (c.f. Chapter 2) which change its status during execution. However, the data contained within these elements are not processed in our approach. As the approach is dealing with compliance context, we are more concerned with the state changes for the data objects as a means of controlling the flow.

1.5 Contributions

The following list summarizes the contributions of this thesis.

- An automated pre-appraisal approach is introduced to provide the aid to SME welling to apply CMMI software improvement model for the configuration management process area.

- A comprehensive syntax and semantics formalization of an excerpt of BPMN elements in Maude is proposed with structural properties (i.e. well-structured and well-formed) that allow for efficient verification procedures.

- A Context-Free Grammar (CFG) for BPMN decision-based gateways guard expressions is introduced in Chapter 3 with a mechanism to evaluate the guards and decide the divergence and convergence of the flow in BPMN models.

- The formalization restricts the semantics of BPMN gateways to the block structure avoiding many structural problems (e.g. deadlocks and lack of synchronization). The gateways block structure makes the merge gateway aware of which split gateway caused the flow to diverge (i.e. using the attribute `itsSplit` and function *Ready2Merge* discussed in Chapter 3), and how many branches had its companion split gateway activated earlier in the process.

- The formalization provides a comprehensive semantics for the data objects, which consider the documents and files that are used as inputs and outputs for different process activities in Chapter 3. Moreover, they are used to control the flow based on their state change. A set of domain-specific rewrite rules and equations are introduced to formally represent data objects state change and creation in the BP.

- A formal proof of soundness of the well-formed BPMN models is introduced in Chapter 4 following the definition of classical soundness in [1].

- A comprehensive LTL formalization of CMMI-CM requirements through mapping into compliance patterns is proposed in Chapter 5. The formalization of the requirement is complete, i.e. all the CMMI-CM requirements are mapped into LTL properties, however, the mapping is not one-to-one relation as explained in Chapter 5.

- A tool support prototype is discussed in Chapter 7 which is designed to support the automation of the approach making it accessible to business related people.

1.6 Structure of the Document

- **(Chapter 1) : Introduction**

  The chapter illustrates our proposed compliance checking approach based on the idea of using model checking technique to test the properties (i.e. requirements). The motivation, contributions, challenges and justification of reasons behind using the tools, languages and techniques in the approach are explained.

- **(Chapter 2) : Preliminaries**

  A brief, yet essential, introduction of the used notations, languages and models

in this thesis for the purpose of making it self-contained. The chapter presents background of BPMN, Maude and CMMI.

- **(Chapter 3) : BPMN Formal Syntax and Semantics**

  This chapter provides a comprehensive formalization of the formal syntax and semantics of an excerpt of the BPMN elements using Maude. This is followed by introducing two structural properties; i.e. well-structured and well-formed BPMN models. The proposed behavioural semantics of BPMN elements is modelled using Maude (possibly conditional) rules and equations simulating the standards in [68] for the well-formed BPMN models.

- **(Chapter 4) : Semantics Verification**

  On verifying the proposed semantics, we discuss the possible deadlock situations and patterns besides the potential lack of synchronization. These concepts are discussed using the notion of gateways block structure and elaborating on how this notion prevents many deadlock situations. In this chapter we provide a formal proof for the soundness of the well-formed BPMN models.

- **(Chapter 5) : Business Process Compliance Checking**

  In this chapter, we apply the proposed semantics of well-formed BPMN models to model check their compliance with the CMMI-CM requirements. These requirements are first mapped into compliance patterns and then into the corresponding LTL formulae. Finally, the properties are model checked and the results are analysed providing potential improvement recommendations.

- **(Chapter 6) : Related Work**

  In this chapter, we introduce the related published work and discuss their relationships with the proposed work in this thesis in three main areas; i.e. BPs formalization and verification, Maude applications for concurrent processes, and related compliance checking approaches.

- **(Chapter 7) : Conclusions and Future Work**

  This chapter contains a summary of the thesis contributions, conclusions and future work.

Chapter 2

Preliminaries

The work presented in this thesis uses some notations and models which need to be introduced first. We use BPs which are built using the Business Process Modelling Notation (BPMN) and then formalize and verify them using Maude. The formalization provides a compliance checking procedures of BPs with CMMI requirements using model checking technique. The chapter is organized as follows: a brief description of the BPMN is introduced in section 2.1, an introduction to Maude is presented in section 2.2, and in section 2.3, the CMMI model is explained focusing on the configuration management.

2.1 BPMN

The Business Process Modelling Notation (BPMN) is a standard notation and model for representing business processes in the design phase of systems development. Designed by the Object Management Group (OMG), BPMN provides standard graphical notations for designing BPs and defining their procedures, relations with other processes, as well as an informal brief execution semantics.

The BPMN 2.0 standard specification [68] defines 50 constructs and their attributes, and according to [63], less than 20 percent of its vocabulary is used regularly in designing BP models. As a result we are going to focus on the main elements of the BPMN based on the metamodel described in Figure 2.1 and the elements graphical representation described in Figure 2.2. The main elements are the flow nodes: activities, events and gateways; the data objects: input, output and stores; connecting flow elements: se-

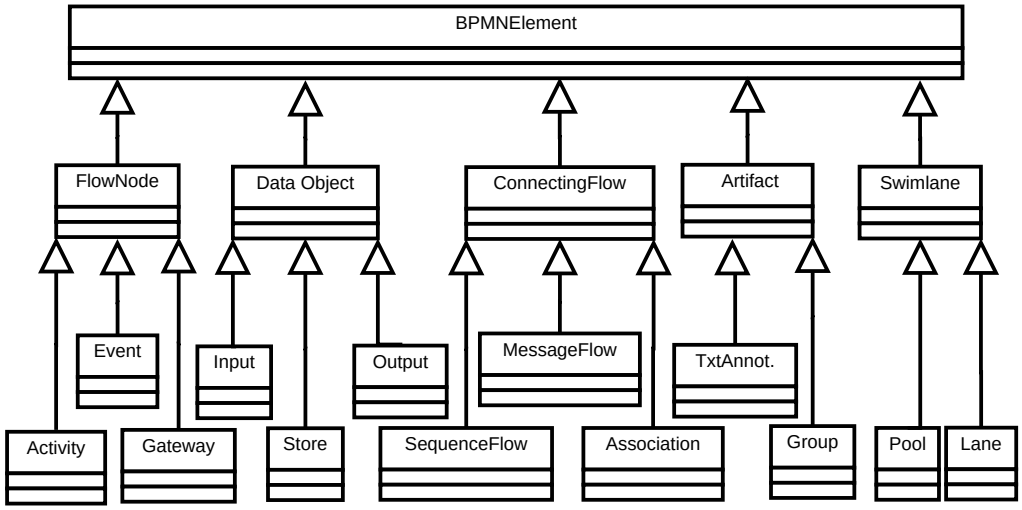


Figure 2.1: BPMN Meta-model

quence flows, message flows and associations; artefacts: groups and text annotations and swimlanes: pools and lanes.

Definition 2.1.1. A BPMN process O is a tuple of sets (OS, T), such that:

- $OS = FO \cup DO$; i.e. OS is the set of flow objects FO and data objects DO,
- $FO = A \cup E \cup G$; i.e. FO is the set of activities A, events E, and gateways G,
- $A = A_{TS} \cup A_{SP}$; i.e. A is the set of tasks A_{TS} and sub-processes A_{SP},
- $E = E_S \cup E_I \cup E_E$; i.e. E is the set of start events E_S, intermediate events E_I and end events E_E,
- $G = G_{AND} \cup G_{XOR} \cup G_{OR}$; i.e. G is the set of AND gateways G_{AND}, XOR gateways G_{XOR}, and OR gateways G_{OR},
- $T = T_S \cup T_M \cup T_{ASSC}$; i.e. T is the set of connecting objects (transitions) of: sequence flow T_S, message flow T_M and associations T_{ASSC}.

An activity is a flow node which represents a process step and is executed automatically or manually [68]. Activities can be tasks or sub-processes. The *task* is the atomic form of an activity, while a *sub-process* is an activity which contains other activities either tasks or other sub-processes, i.e. it can be broken down into a set of activities and

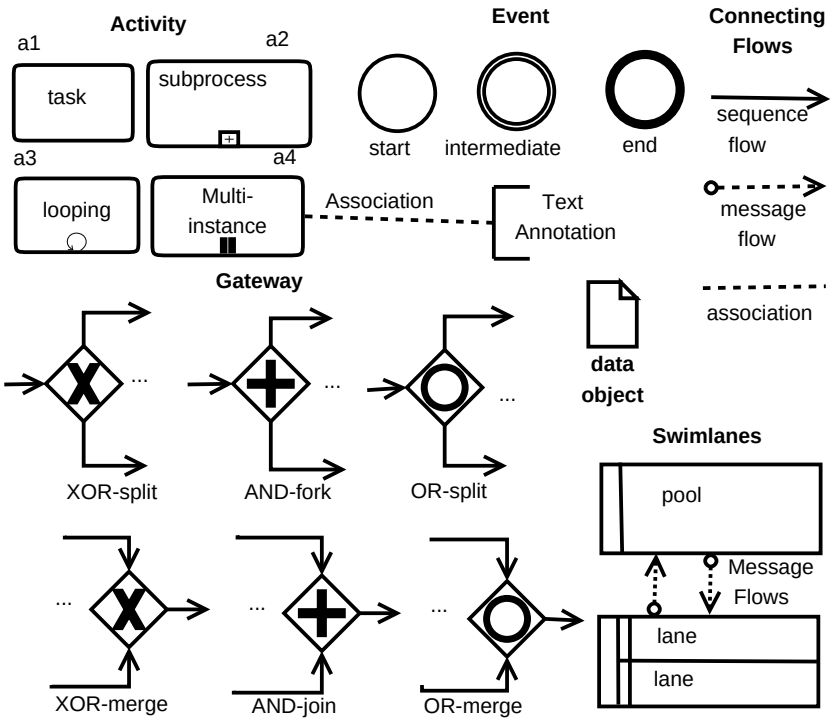


Figure 2.2: BPMN Main Elements

other BPMN elements [68]. Activities can have a marker to indicate its type graphically (e.g. see looping and multi-instance activities in Figure 2.2).

An event is an action that makes changes in the process when it happens. Events normally affect the flow of the process by converting the flow in case of exceptions, initiating a process on receiving a message or even cancelling a process [68]. As defined in Definition 2.1.1, events can occur at the beginning of the process, i.e. start events E_S, at the end of the process, i.e. end events E_E, or in-between the process start and end, i.e. intermediate events E_I (see Figure 2.2). A start event indicates where a particular process starts. A process can be initialized by receiving a message or initialized in a pre-defined date or time. Intermediate events indicate where an event occur somewhere between the start and end of a process. It will affect the flow of the process, but will not start or (directly) terminate the process. BPMN has twelve types of intermediate events : none, message, timer, escalation, error (known as exception), cancel, compensation, conditional, link, signal, multiple, and parallel multiple. One or more intermediate events may be attached directly to the boundary of an activity, e.g. message, timer, exception. In our work, we focus on message and exception intermediate events. Finally, an end event represents where a process terminates and it has no successors.

Gateways are flow nodes that control the divergence and convergence of flows in a process. They determine splitting, forking, merging, or joining of flow paths [68]. The main BPMN gateways are illustrated in Figure 2.2. For example, AND gateways (*ANDgates*) are for parallel execution of more than one branch; data-based exclusive decision XOR gateways (*XORgates*) are for choosing exactly one branch, and inclusive decision OR gateways (*ORgates*) are for choosing one or more branches at the same time. Gateways are either splitting the flow to more than one branch or merging more than one flow branches into one flow.

Data objects carry information and represent documents in the process. This information may represent a singular object or a collection of objects. Data objects are classified as data inputs, data outputs, and data stores. A data input provides the activity with some information, while a data output represents some processed resulting information from the activity which it is linked to. A data store provides a mechanism for activities to retrieve or update stored information that will persist beyond the scope of the process [68].

Artifacts provides the modeller with the capability of showing additional information about a process that is not directly related to the sequence flows or message flows of the process. There are two standard artifacts: groups and text annotations [68]. A group is a grouping of graphical elements that are within the same category for documentation and analysis. The group name appears on the diagram as the group label; e.g. the group `Account Checking` in Figure 2.3. Text annotation is a mechanism for a modeller to provide additional text information for the reader of a BPMN diagram [68]. It is usually connected to other flow objects by the associations.

Connecting flow elements are used to connect all the above mentioned components together to form a BPMN model. Connecting objects can be sequence flows (plain directed arrows connecting events, activity, and gateways together), message flows (dashed arrows with circle attached to its beginning), or associations (dotted arrows). Sequence flows can be: normal, uncontrolled, conditional, default or exception flows. Message flows show the communications through messages between two participants. It connects two separate pools or two activities in two different pools. Associations are connecting the data objects and text annotations to other elements.

Swimlanes are used to group the primary modelling elements through the model. There are two components in a swimlane, pools and lanes. A lane is a sub-partition within a process, sometimes within a pool. They represent the notion of roles and participants in the business process. The communications between different participants are modelled as messages, and this is the only BPMN representation for the means of communications between two different participants (i.e. pools).

As an example of a BPMN model, we use Figure 2.3 which represents a request handling process in a company, where the main participants are the company and the customer. In the company, two departments are involved; i.e. `Accounting` and `Sales`, which are represented as two lanes into the pool `Company`. The messages intermediate events between the company and the customer are: `request` for a customer request, `details` for customer's bank details and `Acknldgmnt` for the acknowledgement from the company includes information about if the company is able to send the product or not. The model shows example activities; such as `Register user` and `request product`, example XOR and AND gateways and example start and end events. The text annotation "`AND join gateway`" marks the AND gateway.

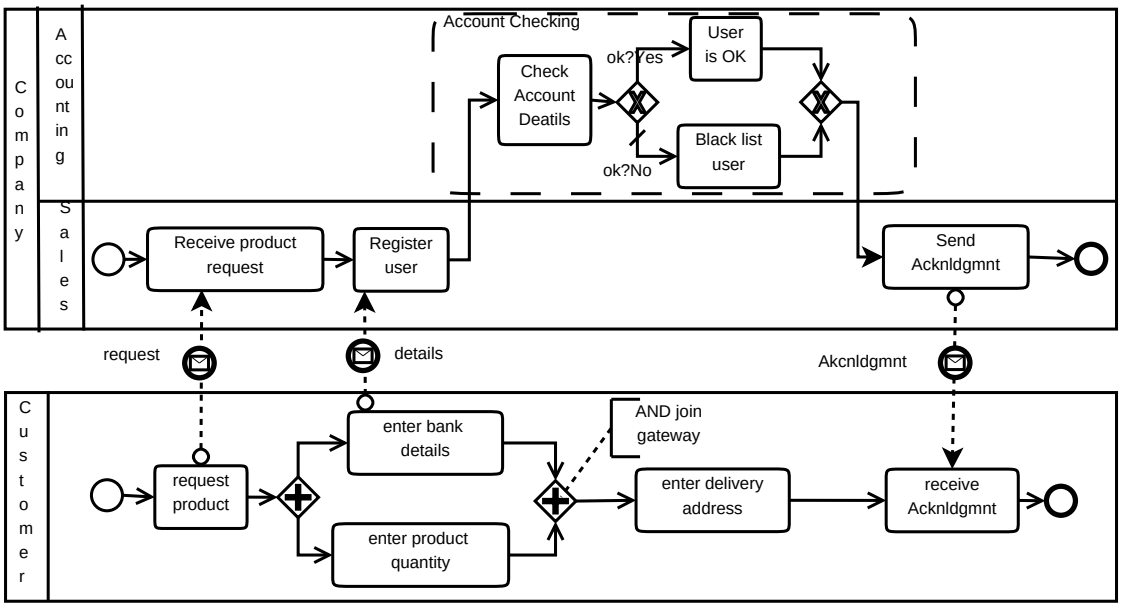


Figure 2.3: BPMN Example

2.2 Maude

Maude [20] is a high-performance term rewriting engine that provides support for both equational and rewriting logic specification and programming of concurrent systems in particular. Based on the definitions in [67], we define the set of terms as the basic formal representation in term rewriting. A term can be a variable, a constant, or a function defined with variables.

Definition 2.2.1. (**Term**): Let $\mathcal{X}$ be a countable set of variables $\{x,y,z,...\}$. Let $\mathcal{F}$ be a signature, i.e. a set of function symbols $\{f,g,...\}$, each having a fixed *arity* given by a mapping $ar : \mathcal{F} \to \mathbb{N}$.
The set of *terms* over the signature $\mathcal{F}$ with variables from $\mathcal{X}$ is the least set $\mathcal{T}(\mathcal{F},\mathcal{X})$ satisfying:
1- if $x \in \mathcal{X}$, then $x \in \mathcal{T}(\mathcal{F},\mathcal{X})$, i.e. a variable is a term,
2- if $a \in \mathcal{F}$ is a constant symbol (i.e. $ar(a) = 0$), then $a \in \mathcal{T}(\mathcal{F},\mathcal{X})$, i.e. a constant is a term,
3- if $f \in \mathcal{F}$ is a *k-ary* function symbol (i.e. $k = ar(f) > 0$) and $t_1,...,t_k \in \mathcal{T}(\mathcal{F},\mathcal{X})$, then $f(t_1,...,t_k) \in \mathcal{T}(\mathcal{F},\mathcal{X})$, i.e. a function of a term is a term.

Let t be a term. $\mathcal{V}ar(t)$ is the set of variables occurring in t and if $\mathcal{V}ar(t) = \emptyset$, then t is ground.

The specifications in Maude are executable logical theories in rewriting logic [56], a logic which forms a flexible framework for expressing a wide range of concurrency models and distributed systems [30]. Maude programs are composed of functional modules and system modules. While functional modules represent theories in membership equational logic, system modules represent theories in rewriting logic. A Maude's functional module is an equational-style functional program with user-definable syntax specifying an equational theory with initial algebra semantics [11]. It specifies a membership equational theory $(\Sigma, Eq \cup U)$ where Σ is the signature of the specification of sorts, subsorts, kinds, and operators in the module, Eq is the collection of statements of equations and memberships(possibly conditional), and U is the set of equational attributes, such as associativity (`assoc`) and commutativity (`comm`) declared for some operators (i.e. extra equations that are treated in special way by the Maude's interpreter to sim-

plify modulo such attributes) [20]. A system module M specifies a rewrite theory $\mathcal{R}$; i.e. $\mathcal{R} = (\Sigma, Eq \cup U, \phi, R)$ where $(\Sigma, Eq \cup U)$ is the membership equational theory specified by the signature equational attributes and equations and membership statements in the module, ϕ is a function that assigns each operator in Σ the number of its frozen (i.e. rewrite with rules is forbidden) arguments, and R is the collection of rewrite rules which may be conditional. A functional module in Maude is declared as `fmod` and a system module as `mod` with the following syntax:

`fmod` ⟨*ModuleName*⟩ `is` ⟨*DeclationsAndStatements*⟩ `endfm`

`mod` ⟨*ModuleName*⟩ `is` ⟨*DeclationsAndStatements*⟩ `endm`

A module can import other modules into it in three different ways; `protecting` (or `pr`), i.e. preserving the sorts and subsorting relations of the imported module, `extending` (or `ex`), i.e. the data of some sort is *extended* with new data elements, yet not identifying previously defined data, or `including` (or `in`), i.e. the imported module is part of the recent module and modifications in its sorts and relations are allowed. In the module definition above, ⟨*ModuleName*⟩ represents the module name which is usually in capitals and ⟨*DeclationsAndStatements*⟩ represents the set of sorting/subsorting relations, operators, equations and rules in the module.

Maude functional modules support multiple sorts, subsort relations in its declarations, as well as operator overloading, and assertions of membership in a sort. The statements in functional modules are the equations and memberships (possible conditional). The conventions in Maude requires module's name to be all capitals, the sort name to start with a capital letter, while the operators start with small letter and each second word starts with a capital letter. An operator is declared with the keyword `op`, its *name*, the list of sorts for its arguments (i.e. the operator's arity or domain sorts), `->`, then the result's sort (i.e. the operator's coarity or range sort), optionally followed by an attribute declaration, followed by a white space and a period.

`op` ⟨*OpName*⟩ `:` ⟨*Sort-1*⟩ ... ⟨*Sort-k*⟩ `->` ⟨*Sort*⟩ `[`⟨*OperatorAttributes*⟩`]` .

Operator attributes (⟨*OperatorAttributes*⟩) provide additional information about the operator and are declared within a single pair of enclosing square brackets, `[` and `]`, after

the sort of the result and before the ending period. The operator attributes [20] are categorized as follows [1]:

1. Equational Attributes: declaring certain kinds of equational axioms for binary operators to facilitate using them by Maude in a built-in way where both domain and range sorts must belong to the same kind, i.e. `assoc` (associativity), `comm` (commutativity), `idem` (idempotency), and `id:Term` (identity, with the corresponding term for the identity element).

2. Constructors: (`ctor`) are the operators appearing in canonical forms (i.e. assuming that the equations in a functional module are (ground) Church-Rosser and terminating[2], then every ground term in the module (that is, every term without variables) will be simplified to a canonical form, perhaps modulo some declared equational attributes).

3. Ditto: (`ditto`) specifies that this operator, being subsort overloaded, should have the same attributes as those appearing explicitly in a previous subsort-overloaded version, except for the `ctor` attribute. See module `SIMPLE-NUMBERS` below for an example (operator `_*_`).

4. Frozen Arguments: (`frozen`) Given a system module `M`, by declaring a given operator, say `f`, as frozen, rewriting with rules is always forbidden in all proper subterms of a term having `f` as its top operator. It is declared as `:` (`op f : S1 ... Sn -> S [frozen]`) specifying that all the arguments of `f` are frozen.

On introducing Maude syntax, we use an example functional module for numbers from [20]. The module `SIMPLE-NUMBERS` defines the addition and multiplication of natural numbers and will be explained eventually in this section. The term defined by an operator can be: a constant if its list of arguments is empty (e.g. `0`), a one argument function (e.g. `s_`), or two arguments function (e.g. `_+_`).

[1] We introduce the attributes that are used later in the proposed syntax and semantics in Chapter 3. The interested reader can refer to [20] for the complete list.

[2] discussed in Section 2.2.2

```
fmod SIMPLE-NUMBERS is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  ops _+_ _*_ : Nat Nat -> Nat [ditto] .
  op _+_ : NzNat Nat -> NzNat [ditto] .
  op _*_ : NzNat NzNat -> NzNat [ditto] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s N + M = s (N + M) .
  sort PostiveNat .
  subsort PostiveNat < Nat .
  cmb N : PostiveNat if N > 0 .
endfm
```

In order to define the behaviour of the plus operator, Maude equations are used. The general form of the unconditional equation is:

eq ⟨Term-1⟩ = ⟨Term-2⟩ [⟨StatementAttributes⟩]

Maude variables can be defined on-the-fly too; i.e. defining the variable where it is being used and not before it in the module. However, the scope of an on-the-fly variable declaration is the declaration's occurrence [20], e.g. the first equation of summation with zero above can be represented as (`eq 0 + X:Nat = X:Nat`). The operation `reduce` is used to reduce the input term. The command (`reduce` {`in` *module* :} *term* .) is used with the possibility not to add the name of the module as it is automatically consider the current module as all the commands which require module name does. This rewriting command causes the specified term to be reduced in many steps using the equations and membership axioms in the given module. The command can be abbreviated to `red` (e.g. `red in SIMPLE-NUMBERS : s s s 0 + s s s s s 0`).

The statements attributes [20] (i.e. ⟨StatementAttributes⟩) represent the attributes associated to module statements defining its features. Maude has four statement attributes; `label`, `metadata`, `nonexec`, and `owise`. The first three can also be used in system modules rules and attribute `metadata` can be used in defining operators.

1. Labels: for tracing, debugging and can be used to name axioms in metalevel. For example, the axiom for idempotency for natural number sets can be labelled in one of the two representations below. While the first statement follows the defined syntax for equations, the second statement uses the general form of labels as introduced in Maude 1 for equations and rules.

```
eq N ; N = N [label natset-idem] .
eq [natset-idem] : N ; N = N .
```

2. Metadata: for attaching string data to the statement as comments about it. In module `SIMPLE-NUMBERS`, the distribution law can be added with the comment documenting it as the distributive law as below:

```
eq (N + M) * I = (N * I) + (M * I) [metadata "distributive law"] .
```

3. Nonexec: for including statements in a module that are ignored by the Maude rewrite engine. A rule can be declared as non-executable using the same attribute in a system module. The above distributive law example can be made non-executable as follows:

```
eq (N + M) * I = (N * I) + (M * I) [nonexec metadata "distributive
law"] .
```

4. Otherwise: for specifying that in all remaining cases, which are not defined by the functions for the same operator, do this statement command. For example, the operator declaration below defines operator `in` for deciding if an object is included into a certain object set. The first equation defines the situation where the object `O` is in the set `A`, giving a `true`. The second equation, which is more general, defines all other cases as `false`. That is, if the first equation does not match, then the object in fact is not in the set, and the predicate should be false.

```
op _in_ : Object ObjectSet -> Bool .
var O : Object .
var A : ObjectSet .
eq O in (O, A) = true .
eq O in A = false [owise] .
```

Equations could be conditional, i.e. limiting its application to certain cases. The general form of the conditional equation in Maude is:

```
ceq ⟨Term-1⟩ = ⟨Term-2⟩ if ⟨EqCondition-1⟩ /\.../\⟨EqCondition-k⟩
      [⟨StatementAttributes⟩]
```

The `EqCondition` has the following concrete syntax for the conditions where `t` and `t'` are terms:

- ordinary equation `t = t'`,
- matching equations `t := t'`, and
- abbreviated Boolean equations of the form `t`, with `t` a term in the kind `[Bool]`, abbreviating the equation `t = true`.

In Maude, sorts are user-defined, while *Kinds* (i.e. error supertypes) are implicitly associated with the connected components of sorts [20]. Kinds are the equivalence classes grouping the sorts which are belonging to the same connected component. Unconditional membership axioms specify terms as having a given sort. This sort must always be in the same kind as that of the term. Conditional membership uses the same `EqConditions` as above, like the conditional equations do. The general form of unconditional and conditional memberships are:

```
mb ⟨Term⟩ :  ⟨Sort⟩ [⟨StatementAttributes⟩]
cmb ⟨Term⟩ :  ⟨Sort⟩ if ⟨EqCondition-1⟩ /\.../\⟨EqCondition-k⟩
      [⟨StatementAttributes⟩]
```

The example `SIMPLE-NUMBERS` above includes a conditional membership. The statement specify that if the natural number is greater than zero, then it is a positive number of sort `PositiveNat`. Note the use of an abbreviated Boolean equation (`N>0`); assuming that the operator (`_>_`) has been defined in the module `SIMPLE-NUMBERS`.

2.2.1 Rewrite Rules

A rewrite theory has an underlying equational theory which contains its declarations (sorts, kinds, and operators) and statements that can be conditional (equations, memberships and rules). Therefore, Maude system modules include the same declarations and

statements as functional modules *plus* the rules. Rewrite rules are the local concurrent transitions for the systems or the logical inference rules [20].

Definition 2.2.2. (**Rewrite Rule**): A rewrite rule r is an ordered pair $l \rightarrow h$, where $l, h \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ are the left- and right-hand sides (*lhs* and *rhs* for short), respectively, and

1- $l \notin \mathcal{X}$, i.e the *lhs* l is not a variable, and

2- $\mathcal{V}ar(h) \subseteq \mathcal{V}ar(l)$, i.e. variables occurring on the h also occur in l.

A rewrite rule r is applicable to term t : $(t \stackrel{r}{\Rightarrow} t')$ if there is a *matching* between the *lhs* pattern of r and a part (or all) of the term being reduced (i.e. t). In this case the pattern in the *rhs* of the rewrite rule r *substitutes* that part (or all) of t in the *lhs* of r producing the new term t'. Maude uses (possibly conditional) rules in the following syntax where the symbols `rl` and `crl` are used for unconditional rewrite rules and conditional rewrite rules respectively.

```
rl [⟨Label⟩] :  ⟨Term-1⟩ => ⟨Term-2⟩ [⟨StatementAttributes⟩]
crl [⟨Label⟩] :  ⟨Term-1⟩ => ⟨Term-2⟩ if ⟨Condition-1⟩ /\.../\
      ⟨Condition-k⟩ [⟨StatementAttributes⟩]
```

The conditions in Maude's conditional rule (i.e. `Condition`) are more general than the equation conditions (`EqCondition`) as it can include rewrite expressions with syntax $(t \Rightarrow t')$ besides equations and memberships. However, like the `EqCondition` equations, the equations in the rules conditions can be matching or abbreviated Boolean equations.

In the following we use example Maude system module `SIMPLE-VM` [20] to represent a vending machine which a user inserts a coin (i.e. a quarter or a dollar) and the machine returns an item `a` and it can return change for the user as the item `a` priced at three quarters.

```
mod SIMPLE-VM is
   sorts Coin Item Marking .
   subsorts Coin Item < Marking .
   op null : -> Marking .
   op __ : Marking Marking -> Marking [assoc comm id: null] .
   op $ : -> Coin .
   op q : -> Coin .
   op a : -> Item .
```

```
    var M : Marking .
    rl [add-q] : M => M q .
    rl [add-$] : M => M $ .
    rl [buy-a] : $ => a q .
    rl [change] : q q q q => $ .
endm
```

The behaviour is simulated using the rules with labels `add-q` for inserting a quarter, `add-$` for inserting a dollar, `buy-a` for retrieving an item and a quarter, and `change` for computing the change of 4 quarters as one dollar. The rewriting command (`rewrite` {`[` *bound* `]`} {`in` *module* :} *term*) causes the specified term to be rewritten using the rules, equations, and membership axioms in the given module. Maude's interpreter applies the rules (if no equation can be applied) using rule-fair top-down (lazy) strategy and stops when the number of rule applications reaches the given bound [20]. If the upper bound clause is omitted, infinity is assumed. The command may be abbreviated to `rew` as shown in the rewrite command for the vending machine example here.

```
Maude> rew [2] $ $ q q .
rewrite [2] in SIMPLE-VM : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rew/sec)
result Marking: $ $ $ q q q
```

In order to explain how Maude obtained the result above, we use one of the debugging and optimizing Maude programs approaches, i.e. tracing. The tracing facilities allow us to follow the execution of our specifications, that is, the sequence of rewrites or equational simplification reductions that take place. It should be turned on first with the command `set trace on .` Then the above rewrite command will result in:

```
Maude> rew [2] $ $ q q .
rewrite [2] in SIMPLE-VM : $ $ q q .
********** rule
rl M => q M [label add-q] .
M --> $ $ q q
$ $ q q
```

```
--->
q $ $ q q
********** rule
rl M => $ M [label add-$] .
M --> $ $ q q q
$ $ q q q
--->
$ $ $ q q q
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Marking: $ $ $ q q q
```

The configuration `$ $ q q` is rewritten into `$ $ $ q q q` in two rewrite steps. The first is by applying rewrite rule `add-q` and the second is by applying rewrite rule `add-$` as shown in the code above. The tracing can be switched off using command `set trace off .`

2.2.2 Admissible Modules

In Maude modules, the equations must satisfy the requirements of being Church-Rosser, terminating, and sort decreasing. This is because the computation is accomplished by using the equations as rewrite rules until a canonical form is found. As a result it guarantees that all terms in an equivalence class modulo the equations will rewrite to a unique canonical form, and that this canonical form can be assigned a sort that is smaller than all other sorts assignable to terms in the class. To discuss the idea of admissible modules, we introduce the basic properties based on the definitions in [20], e.g. confluence, termination and Church-Rosser. A set of equations Eq is *confluent* when any two rewritings of a term can always be unified by further rewriting: if $t \rightarrow_E^* t_1$ and $t \rightarrow_{Eq}^* t_2$, then there exists a term t such that $t_1 \rightarrow_{Eq}^* t'$ and $t_2 \rightarrow_{Eq}^* t'$. A set of equations Eq is *terminating* when there is no infinite sequence of rewriting steps $t_0 \rightarrow_{Eq} t_1 \rightarrow_{Eq} t_2 \rightarrow_{Eq} \ldots$. If E is both confluent and terminating, a term t can be reduced to a unique canonical form $t \downarrow_{Eq}$, i.e. to a unique term that can no longer be rewritten. A set of equations Eq is Church-Rosser if it is confluent.

All conditional equations $t = t'$ *if* $C_1 \wedge \ldots \wedge C_n$ in a functional module `M` have to

satisfy the following admissibility requirements, (ensuring that all the extra variables will become instantiated by matching):

1. $vars(t') \subseteq vars(t) \cup \bigcup_{j=1}^{n} vars(C_j)$.
2. If C_i is an equation $u_i = u'_i$ or a membership $u_i : s$, then $vars(C_i) \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(C_j)$.
3. If C_i is a matching equation $u_i := u'_i$, then u_i is an `M`-pattern and $vars(u'_i) \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(C_j)$.

The modules built for formalizing the syntax and semantics of BPMN models are admissible as they are terminating and Church-Rosser. This result is based on testing the modules using the Maude verification toolkit, i.e. Termination Tool and Church-Rosser Tool.

2.2.3 Model Checking

BPs as concurrent systems often require environment interactions and continuous execution with possible successful termination. One of the most popular techniques for reasoning about concurrent software systems and debugging is *model checking*. Recalling the strengths of model checking mentioned in Chapter 1, model checking is considered a powerful candidate for the compliance problem over other verification techniques due to the criticality of the systems that it can handle, the expressiveness of the used formal languages and the automation [9]. Moreover, LTL model checker is part of the Maude package [31] which makes it easier and straightforward to feed the model checker with our Maude formalization for the model under consideration.

In [19], model checking is a collection of automatic techniques used to verify finite state concurrent systems. It contains basically three main components; (1) a model specification language used to build the system formal description as a finite-state transition system, (2) a property specification language (normally based on a temporal logic) is used to build the system properties which are needed to be checked, and (3) a verification procedure, i.e. exhaustive searching of the model state space in order to decide whether the specified property is satisfied or not.

Some model checker verification procedures search the state space for the negation of the property specified (e.g. [97]) and others search for the exact property specified which requires the user to enter the property in the negation form (e.g. [31]). The idea is to search for the unwanted trace of transitions which violate the property. The algorithm, in [97], assumes that the specification is an LTL formula p. It concluded that checking that the system satisfies that formula is equivalent to checking that it satisfies its negation $\neg p$. An automaton $B_{\neg p}$ that accepts the traces that satisfy $\neg p$ is constructed. Then, this automaton is composed with an automaton that accepts the traces of the system model (M). If the composition is empty, then M satisfies the specification p. Otherwise, any of the traces recognized by the composite automaton is a *counterexample*. A counterexample is a trace of the process execution that does not satisfy the checked property. On the other hand, in on-the-fly LTL model checking, a Büchi automaton is constructed from the negation of the property formula and then lazily searching the synchronous product of the Büchi automaton and the system state transition diagram (in Kripke structure) for a reachable accepting cycle [31] using depth-first search. If no accepting runs exist, we can conclude that every initial state satisfies the specification.

For a rewriting theory $\mathcal{R}$, which represents a BPMN model, Maude's LTL model checker associates a Kripke structure $\mathcal{K}$ if the modules are Church-Rosser and terminating. Kripke structure is a state transition graph which represent the computations of systems (i.e. infinite sequence of states where each state is obtained from the previous state by some transition [46]). They are the natural models for propositional temporal logic [20]. Then, the model checker solves a satisfaction problem of the form

$$\mathcal{K}(\mathcal{R}, k)_{\Pi}, [t] \models \varphi \tag{2.1}$$

where k is a kind of the states from the rewrite theory $\mathcal{R}$, Π defines the state predicates, $[t]$ a kind of initial states, k is the kind of model states, and φ is the property to be checked. The output of the model checking is either *true*, i.e. the property is satisfied, or a *counterexample*, i.e. the property is not satisfied.

2.3 CMMI

Capability Maturity Model Integration (CMMI) is a guide to implement a continuous process improvement for developing products and services [24]. The CMMI is developed by SEI-CMU to provide a collection of systematic high level descriptive best practices that can be used as a reference model for software development small and medium-size enterprises (SME) in their process improvement cycle [21]. A SME applies such process improvement approach to improve the quality of the developed software, by following a well-organized reference process.

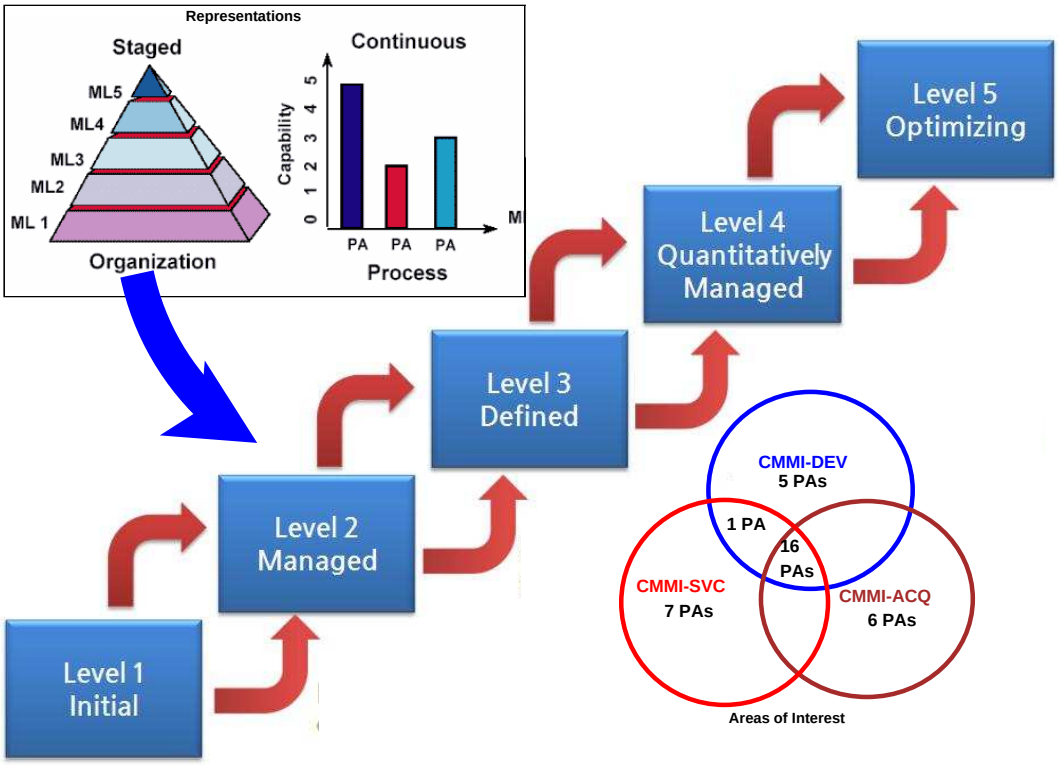


Figure 2.4: CMMI representations, maturity levels and areas of interest

CMMI provides two representations: continuous and staged as illustrated in Figure 2.4. The *staged* representation (in Figure 2.4) assesses the maturity level of a whole development process across multiple process areas and hence uses the *maturity* levels. There are five maturity levels numbered 1 through 5 (i.e. ML1: Initial, ML2: Managed, ML3: Defined, ML4: Quantitatively Managed, and ML5: Optimizing). The *continuous* representation assesses the capability level of individual Process Areas (PAs) that

are selected based on the organization's business goals and hence uses *capability* levels. There are four capability levels numbered 0 through 3 (i.e. CL0: Incomplete, CL1: Performed, CL2: Managed, CL3: Defined). In the continuous representation, the process areas are rated individually. Both capability levels and maturity levels help to improve the organization processes and measure how well it can (and do) improve their processes. However, [21] claims that experience has shown that organizations do their best when they focus their process improvement efforts on a manageable number of process areas at a time (i.e. staged representation). Hence we were encouraged to focus on the staged representation of the CMMI as a basis for the approach in this thesis. According to the CMMI [22], the following is a brief idea about the staged representation maturity levels.

- ML1 (*Initial*): processes are usually ad hoc and chaotic with no support environment. The success depends on the people and not a stable process with inability to repeat the success.

- ML2 (*Managed*): processes are planned and executed in accordance with policy employing skilled relevant people and the existing practices are retained during times of stress using milestones (possibly for a specific project).

- ML3 (*Defined*): processes are well characterized and understood, and are described in standards, procedures, tools, and methods. A project's standards and procedures are tailored from the organization's set of standard processes.

- ML4 (*Quantitatively Managed*): the organization and projects establish quantitative objectives for quality and process performance and use them as criteria in managing projects and assessing outcomes.

- ML5 (*Optimizing*): an organization continually improves its processes based on a quantitative understanding of its business objectives and performance needs.

A Process Area (PA) is a cluster of related practices in an area that, when implemented collectively, satisfies a set of goals considered important for making improvement in that area [21]. For each maturity level, there is a number of process areas that describe the best practices related to it. A summary of the relations among the CMMI components in the staged representation we follow here is presented in Figure

2.5 (adapted from [4]). The PAs can be classified into four basic categories for its areas of impact on the company's business process. These categories are: *process management, project management, engineering* and *support*. They allow the process elements to have relationships among each others for integration, such as sub-practices in one PA that affects another PA. For example, the Decision Analysis and Resolution (DAR) process area (i.e. a support process area at ML3) contains specific practices that address the formal evaluation process used in the Technical Solution (TS) process area (i.e. an engineering process area at ML3) for selecting a technical solution from alternative solutions.

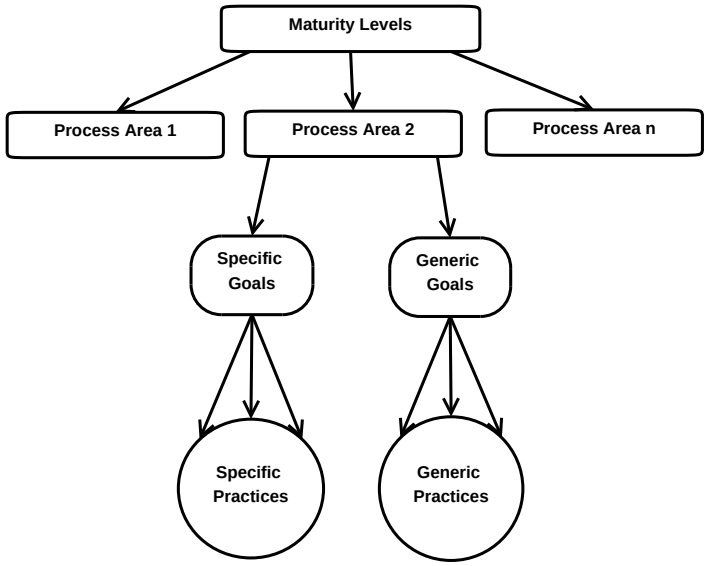


Figure 2.5: CMMI Staged Representation Structure

CMMI has three basic areas of interest into BPs (c.f. Figure 2.4); CMMI for Development, CMMI for Services, CMMI for Acquisition. While CMMI for Development (CMMI-DEV) focuses on product and service development processes, CMMI for Acquisition (CMMI-ACQ) focuses on supply chain management, acquisition, and outsourcing processes in government and industry, and CMMI for Services (CMMI-SVC) focuses on delivering services within an organization and to external customers. In this work, CMMI-DEV [21] is used as the reference model. The CMMI-DEV (referred to as CMMI in the rest of the work) contains 22 process areas indicating the aspects of product development that are to be covered by company processes[3]. Table 2.1 lists the process areas

[3]16 process areas out of the 22 process areas in the CMMI-DEV are common among the three areas

(PAs) in each maturity level (ML) and its category.

Table 2.1: CMMI-DEV Process Areas

Process Area (PA)	**ML**	**Category**
Configuration Management (CM)	ML2	Support
Measurement and Analysis (MA)	ML2	Support
Project Monitoring and Control (PMC)	ML2	Project Management
Project Planning (PP)	ML2	Project Management
Process and Product Quality Assurance (PPQA)	ML2	Support
Requirements Management (REQM)	ML2	Project Management
Supplier Agreement Management (SAM)	ML2	Project Management
Decision Analysis and Resolution (DAR)	ML3	Support
Integrated Project Management (IPM)	ML3	Project Management
Organizational Process Definition (OPD)	ML3	Process Management
Organizational Process Focus (OPF)	ML3	Process Management
Organizational Training (OT)	ML3	Process Management
Product Integration (PI)	ML3	Engineering
Requirements Development (RD)	ML3	Engineering
Risk Management (RSKM)	ML3	Project Management
Technical Solution (TS)	ML3	Engineering
Validation (VAL)	ML3	Engineering
Verification (VER)	ML3	Engineering
Organizational Process Performance (OPP)	ML4	Process Management
Quantitative Project Management (QPM)	ML4	Project Management
Causal Analysis and Resolution (CAR)	ML5	Support
Organizational Performance Management (OPM)	ML5	Support

CMMI uses the term *Institutionalization* for ingraining the process in the way the work is performed and there is commitment and consistency to perform (i.e. execute) the process [21]. This is a way of making the designed BP a true representation of what activities are conducted in real implementation. In case that requirements and objectives

of interest as illustrated in Figure 2.4

for the process have changed, the implementation of the process may also need to change to ensure that it remains effective. The generic practices describe activities that address these aspects of institutionalization and the generic goals reflect this. Hence, the generic goals (GG) are associated with a certain level of process progression [22] (i.e. maturity level). While achieving the GG1 indicates a performed process, achieving GG2 indicates a managed process and GG3 indicates a defined process.

2.3.1 Configuration Management

Configuration Management (CM) is a support process area at ML2. CM is defined in [21] as a discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements[4]. The CM importance appears in controlling quality, cost and schedule of the organization's products (or services) throughout its life cycle. It includes three main activities: (1) identification of the product characteristics, (2) control of changes to those characteristics, and (3) recording and reporting on change processing and implementation status. Processes and tools used to successfully complete these three activities in any environment (e.g. CMMI [22], ITIL [50], COBIT [101], ISO [45]) can be subdivided into five traditional CM functions which have specific configuration purposes: (1) planning, (2) identification, (3) control, (4) status accounting, (5) verification and audit.

All the components that are used to deliver a company's product or service are considered configuration items (CIs). According to the CMMI, the work products placed under configuration management (i.e. CIs) include the products that are delivered to the customer, designated internal work products, acquired products, tools, and other items used in creating and describing these work products. However, based on our observation, despite having similar configuration management procedures, each organization has their own customized definition for their CIs. The CIs are baselined in order to

[4]There are more than 900 definitions and taxonomies for CM types on the IF4IT on (`http://www.if4it.com/SYNTHESIZED/FRAMEWORKS/TAXONOMY/configuration_management_taxonomy.html`). However, we will use the CMMI definitions here.

be used in the development process. A baseline [21] is a set of specifications or work products that has been formally reviewed and agreed on, which thereafter serves as the basis for further development, and which can be changed only through change control procedures. The CM process area consists of three specific goals under which seven specific practices are defined as detailed in Appendix C. They define a number of best practices that a CM should have in order to be compliant with the CMMI. The three specific goals are: (1) establish baselines for the used CIs, (2) track and control changes, and (3) establish integrity for records and audits.

2.3.2 CMMI Appraisals

A company should be qualified to a certain level of maturity if its process satisfies all the related requirements in the process areas belong to it. The evaluation (i.e. appraisal) is done by the SEI itself or one of its trained partners worldwide and the results are centrally published on their website[5]. According to the CMMI definition [21], an appraisal is an examination of one or more processes by a trained team of professionals using an appraisal reference model (i.e. CMMI here) as the basis for determining, at a minimum, strengths and weaknesses. An organization cannot be certified in CMMI; instead, an organization is *appraised*. Depending on the type of the appraisal, the organization can be awarded a maturity level rating or a capability level achievement profile. The appraisal is considered as an indicator of how well the company's processes compare to CMMI best practices, and important to identify areas where improvement can be made in the process. Moreover, external customers and suppliers will be aware of how well the organization's processes compare to CMMI best practices, which allow for serious contracts and collaborations, and some contractual requirements of one or more customers may contains compliance to CMMI as a condition.

Appraisals of organizations using a CMMI model must conform to the requirements defined in the Appraisal Requirements for CMMI (ARC) document [21]. There are three classes of a CMMI appraisal, i.e. A, B and C. A class A appraisal is expected to be the most accurate, designed to maximize buy-in from the appraisal participants, and offers the organization the best understanding of its issues that need to be fixed and its strengths

[5] https://sas.cmmiinstitute.com/pars/

that should be shared [60]. Class B describes a smaller scale appraisal methodology, sometimes called a mini-appraisal [60], which can be accomplished with a smaller team of expert appraisers over a reduced number of days. It can be used to spot-check the organization between full appraisals. Class C describes the least intensive appraisal methodology, sometimes called a micro-appraisal or questionnaire-based appraisal. A class C appraisal can be used to get a rough idea of the current state of the practice within an organization [60]. The characteristics of the CMMI appraisal classes are summarized in Table 2.2 which is adapted from [60].

The Standard CMMI Appraisal Method for Process Improvement (SCAMPI) describes a class A appraisal method [83]. The SCAMPI is designed to provide benchmark-quality ratings relative to CMMI models [83]. SCAMPI method deals with the consolidation of evidences (e.g. presentations, documents and interviews) related to the execution of the process in actual projects [24]. The assessment team uses these evidences to support the rating of practices, goals and, hence, to evaluate the PAs. However, an appraisal is more expensive for a SME than for a larger one. These costs are usually measured by considering the cost spent on appraisal related training, approach verification using SCAMPI Class C, deployment verification using SCAMPI Class B, and institutionalization verification using SCAMPI Class A appraisals divided by the number of associates in the organizational unit per appraisal period [66]. They can be in the form of hiring expert lead appraisers, and spending massive working hours in staff interviews and internal (and/or external) appraisal team meetings. Here, there is a need to reduce the cost of the appraisals as they are being performed every two or three years. One way of doing this is to use less expensive methods to follow a *pre-appraisal* formal procedure aiming at saving the money, time and effort. Although the social cultural concerns are not included in the scope of this thesis, this will not eliminate the need to spread the CMMI culture among the organization staff in its early stages by the managers.

In our proposed approach, we present a semi-automatic compliance checking method for the CM process area. We believe that this approach can be used as a *pre-appraisal* formal method to check how ready the designed business process is to go for the expensive appraisal methods. The method we present uses the designed BP process and check its compliance with the CMMI-CM practices. Based on this check results, a designed

Table 2.2: Characteristics of CMMI appraisals and our approach

Feature	**Class A**	**Class B**	**Class C**
Usage Mode	In-depth investigation Basis for improvement	Self appraisal	Quick-look
Advantages	Strengths and Weaknesses of PAs Robust method with Consistent, repeatable results	A starting point focuses on areas that need most attention	Inexpensive, rapid feedback Short duration
Disadvantages	Demands significant resources	Not used for rating No deep coverage	Not used for rating Less ownership of results
Sponsor	Senior Manager	Any Manager	Any Internal Manager
Team Size	4-10 and ATL[a]	1-6 and ATL	1-2 and ATL
Team Composition	External and internal	External or internal	External or internal

[a] ATL: Appraisal Team Leader.

process can be judged as a CMMI-CM compliant or some certain practices need to be improved, with initial information about which properties are not satisfied to look for the possible improvement in the process. Assuming that the SME is following their designed CM process (or the designed process reflects what is actually being done in the SME), the proposed method will allow the SME to start building their stable CM process which can be ready for more advanced form of appraisal (i.e. SCAMPI A, B, or C).

2.4 Chapter Summary

In this chapter, the basic notions and tools used in this thesis are introduced. First the BPMN notation for BPs is presented in Section 2.1, then in Section 2.2, the Maude language is introduced. After that a brief introduction is given for the CMMI; i.e. its contents, appraisal method with focusing on the CM process area as the application area for the proposed approach in this thesis in Section 2.3. Next chapter will present the syntax and semantics of BPMN models using Maude language.

Chapter 3

BPMN Formal Syntax and Semantics

BPMN elements have been formally mapped into many formal languages, e.g. Petri nets [27], YAWL [91], and CSP [104]. However, as explained in Chapter 6, most of the formalizations do not provide a comprehensive formalization approach for handling data objects, guard expressions, and possibility of deadlocks related to the decision based gateways. In this chapter, we provide the details of the BPMN2MAUDE procedure illustrated in Figure 1.1 where the formal syntax and semantics of an excerpt of the BPMN elements is introduced using Maude. In the first part of this chapter, we present a formal syntax in Maude for the BPMN 2.0 core elements; flow nodes (i.e. activities, events, and gateways), connecting flow (i.e. sequence flows, message flows, and associations), data objects (i.e. input, output and data stores), swimlanes (i.e. pools and lanes), and artefacts (i.e. groups and text annotations) with a focus on the gateways structure, guard representation and data objects. The notion of well-formed BPMN process models is introduced and formalized to allow for formal sound models as will be discussed in the next chapter. In the second part of this chapter, the behavioural semantics of BPMN elements is modelled using Maude (possibly conditional) rules and equations mapping the behaviour standards in [68] for the well-formed BPMN models.

BPMN 2.0 has five main categories of elements: flow nodes, connecting flow, swimlanes, data and artefacts according to the metamodel in Figure 2.1. These elements are dependently defined and used as each one of them should be connected to one (or more) other elements in order to build the BPMN model (or diagram). The main flow elements are modelled as sorts `FlowNode`, `DataObject`, `ConnectingFlow`, `Artifact`, and `Swimlane` while the object's subsorts are defined as `DataStore`, `DataInput`, `DataOutput`, `Activity`, `Event`, `Gateway`, `SequenceFlow`, `MessageFlow`, `Association`, `Group`, `TxtAnnotation`, `Pool`, and `Lane`. An activity is a flow node, but not all flow nodes are activities and a message flow is a connecting flow but not all connecting flows are sequence flows. Therefore the subsorting relations should be established[1].

```
subsorts DataInput DataOutput DataStore < DataObject .
subsorts Activity Event Gateway < FlowNode .
subsorts Pool Lane < Swimlane .
subsorts SequenceFlow MessageFlow Association < ConnectingFlow .
subsorts TxtAnnotation Group < Artifact .
subsorts FlowNode ConnectingFlow Swimlane DataObject Artefact
         < FlowElement .
```

At the same time, these elements represent the objects types in our main configuration, i.e. the `FlowElement` is a subsort of sort `object`. The set of objects that represent the BPMN elements in a BPMN process is modelled as `ObjectSet` as the BPMN process is a set of BPMN Flow elements.

```
subsorts FlowElement < Object < ObjectSet .
```

It is worth saying that Maude appreciates the white spaces between any two words in the language, e.g. there should be a space before the period at the end of a statement, between an operator fixed sub-term and a variable (e.g. `t_` in definition is used as `t N` in the semantics). Most of these spaces will be ignored in this chapter. However, the full working Maude code is available in the attached code file (See Appendix

[1]Notice that the swimlanes are not modelled as separated objects in this formalization. Instead they are represented as attributes in their corresponding flow elements objects as descried in Section 3.1.5.

A for details). Basically, the flow nodes are represented as objects in the general configuration (`< Oid : Cid | AS >`), where `Oid` represents the object identifier, `Cid` represents the corresponding class identifier, and `AS` represents the set of object attributes. Object identifiers are represented by the symbols `a`*i* for the activities, `e`*i* for the events, `g`*i* for gateways, and `t`*i* for flow transitions (defined as `op t_:Nat->TransSymbol` with `op notrans:->TransSymbol`)[2], where $i \in \mathbb{N}$. The symbols `m`*i*, `d`*i* and `tann`*i* are used for messages, data objects and text annotations respectively. Therefore, the following subsorting relation holds in the specifications.

```
subsorts ActivitySymbol EventSymbol GateSymbol FlowOid
         MsgSymbol DataSymbol TextAnnotationSymbol < Oid .
```

Notice that in Maude representation of these symbols a space should be used between the symbol and the number (i.e. `a i` instead of `a`*i* used here in the text for presentation purposes). The class identifier represents object's type (sort) by using pre-defined operators of the main sorts; (`task`) for task activity, (`subprocess`) for sub-process activity, (`aforkgate` and `ajoingate`) for AND fork and join gateways, (`xsplitgate` and `xmergegate`) for XOR split and merge gateways, and (`osplitgate` and `omergegate`) for OR split and merge gateways respectively. The set of attributes (`AS`) represents the specific properties of the object assigned with corresponding values. For example, attributes (`name`,`in` and `out`)[3] contain the object's name, and transition identifiers for the input and output flows which connects the object with its predecessors and successors respectively. Therefore, the relationships between objects are implicitly represented into the objects attributes (`in` and `out`) which determine the exact place of an object with respect to other objects in the process (i.e. its immediate predecessors and successors). The attributes are defined as operators which are assigned transition identifier(s) indicating the object predecessors or successors.

```
op name':_ : String -> Attribute .
op in':_ : TransSymbol -> Attribute .
op out':_ : TransSymbol -> Attribute .
```

[2]The full Maude modules are included into the attached code file (See Appendix A for details).

[3]Similar attributes are used in several approaches (e.g. [6, 104, 102])

In the following sections a detailed description of the BPMN elements associated with the proposed formal syntax mapping from BPMN elements into Maude.

3.1 BPMN Flow Elements

3.1.1 Activities

Activities are represented as objects of the form `< a i : ActivityCid | AS >`, where `a i` is the object identifier, `ActivityCid` is the activity type. A task can be of type `send`, i.e. to send a message to an external participant, `receive`, i.e. to wait for a message to arrive from an external participant, `user`, i.e. a human performs the task with the assistance of a software application, `manual`, i.e. performed without the aid of any BP execution engines or any application, and `service`, i.e. uses a web service or an automated application [68]. The task type is represented as an attribute and its values as operators as described below.

```
op taskType‘:_ : TaskType -> Attribute .
ops send receive user manual service : -> TaskType .
```

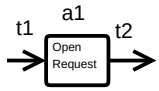
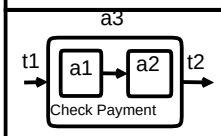
BPMN Activity	Maude Representation
	**< a 1 : task | name : "Open Request" ; taskType : user ; in : t 1 ; out : t 2 ; cond : false ; ToBeActive : false ; active : true ; hasInput : d 1 >**
	**< a 3 : subprocess | name : "Check Payment" ; in : t 1 ; out : t 2 ; contains : (a 1,a 2) ; cond : true ; active : false ; ToBeActive : false >**

Figure 3.1: Mapping from BPMN Activities to Maude Representation

In our formalization, we use the same notion of activity markers as attribute `marker` giving further details of its type. We define the operators `loop` and `MI` to represent a looping activity and a multi-instance activity respectively. The attributes of the activity will determine if it is repeated or performed once. A looping activity has the attribute `loopingno`, and a multi-instance activity has the attribute `instances` which is assigned

a Natural number value for each one of them indicating the number of loops it should make.

```
op marker‘:_ : Marker -> Attribute .
ops loop MI AdHoc Compensation : -> Marker .
op loopingno‘:_ : Nat -> Attribute .
op instances‘:_ : Nat -> Attribute .
```

An example of a `task` object in Figure 3.1 is `a1` as the activity symbol, `task` is its sort type, and `name`, `in`, and `out` are attributes representing the object name (`Open Request`), its incoming transitions `t1` and outgoing transitions `t2` respectively. As a sub-process can be decomposed into concrete tasks and other BPMN elements, the attribute `contains` is used to represent the activities in it. It contains a reference to the set of objects that contains the detailed elements in the sub-process, which in turn can contain sub-processes. An example of a sub-process can be found in Figure 3.1 where attribute `contains` represents the sub-process contents `a1` and `a2` which are other objects in the same process (defined as `op contains‘:_ : Oid -> Attribute`).

3.1.2 Events

An event is represented as an object of the form: `< e i : EventCid | AS >`, where `EventCid` represents the type of the event. Following Definition 2.1.1, events can be `start` to initiate a process, `end` to indicate process completion or `intermediate` for triggering certain kinds of behaviour during the process, such as exceptions and messages. The operators `startEvent, intermediateEvent, endEvent` defines the three types. The symbols o_s and o_e represents the start and end events respectively.

```
ops start end exception message : -> TypeofEvent .
ops startEvent intermediateEvent endEvent : -> EventCid [ctor] .
op eventType‘:_ : TypeofEvent -> Attribute .
```

An example mapping from BPMN events to the corresponding Maude representation is given in Figure 3.2 where `e1` is a start event (i.e. a plain start in the first row and message start in second row), `e2` is an end event, `e3` is an intermediate boundary-attached error event (exception) and `m1` is an intermediate message event between two pools (i.e.

participants) in a process. The start event can be of type plain `start`, `message` or `timer`. Each type of the three event types can have different subtypes, the attribute `eventType` is declaring the specific type of the event. We are using `notrans` to represent the "no incoming flow" situation.

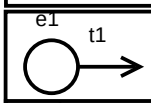
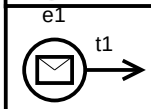
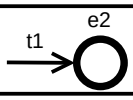
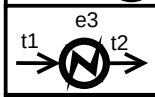
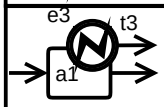
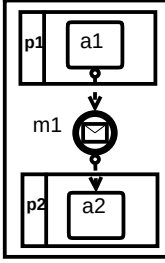
BPMN Event	Maude Representation
	**< e 1 : startEvent | eventType : start ; in : notrans ; out : t 1 ; process : true ; cond : false ; ToBeActive : false ; active : false >**
	**< e 1 : startEvent | eventType : message ; in : notrans ; out : t 1 ; process : false ; cond : true ; ToBeActive : false ; active : false >**
	**< e 2 : endEvent | eventType : end ; in : t 1 ; out : notrans ; cond : false ; ToBeActive : false ; active : false >**
	**< e 3 : intermediateEvent | eventType : exception ; in : t 1 ; out : t 2 ; boundary : false ; cond : false ; ToBeActive : false ; active : false >**
	**< e 3 : intermediateEvent | eventType : exception ; in : notrans ; out : t 3 ; linkedObject : a 1 ; boundary : true ; cond : true ; ToBeActive : false ; active : false >**
	**< m 1 : intermediateEvent | eventType : message ; sourceObject : a 1 ; sourcePool : "p1" ; targetPool : "p2" ; targetObject : a 2 ; messageInfo : "Infromation" ; cond : false ; ToBeActive : false ; active : false >**

Figure 3.2: Mapping from BPMN Events to Maude Representation

For the intermediate events, recall that there are twelve types of them in BPMN. We will focus on `message` and `error` (i.e. exception) in this formalization. Exceptions can occur as part of the gateway behaviour if no flow is available to pass the activation to, as will be explained in Section 3.4. An intermediate event of type error (exception, as will be called afterwords) is represented as an object with attribute `eventType` with value `exception` (e.g. in Figure 3.2 `e3` in the fifth row). Attribute `linkedObject` represents the object which the exception is attached to its boundary. An intermediate event can be in between other objects (i.e. `e3` in the sixth row in Figure 3.2), where it has an incoming flow from its predecessor object.

Messages may be connected to the pool boundary or to a flow object within the pool boundary. However, they do not connect two objects within the same pool. The sym-

bol `mi` represents the message identifier. The message has the attributes `sourcePool` and `targetPool` representing the message source and target participants respectively. The message can carry information which the modeller want to specify at design time in the attribute `messageInfo`. If the message is sent by a certain activity in the source pool to a certain activity in the target pool, then it should be specified using attributes `sourceObject`, and `targetObject` respectively. An example message object is message `m1` in the last row of Figure 3.2. Finally, an end event can be of type `end` indicating the termination point in the process. An example for an end event is event `e2` in the third row of Figure 3.2.

3.1.3 Gateways

In our formalization, gateways are represented as objects of the form: `< g i : GateCid | AS >`, where `GateCid` represents the type of the gateway to be one of the (`aforkgate, ajoingate, xsplitgate, xmergegate, osplitgate, omergegate`) for AND fork and join, XOR split and merge, and OR split and merge respectively.

AND *fork* gateway divides a path into two or more parallel paths which can be performed concurrently, rather than sequentially while *join* gateway combines two or more parallel paths into one path (e.g. synchronizer). We use *aforkgate* and *ajoingate* for AND fork and join gate respectively. Examples of a parallel fork and a parallel join are `g1` and `g4` respectively in Figure 3.3 where the outgoing flows from `g1` are `t2` and `t3` and the incoming flows to `g4` are `t2` and `t3`.

A diverging decision-based exclusive gateway (XOR split) is used to create alternative paths within a process flow based on conditional expressions contained within the outgoing sequence flows [68], where only one of the alternatives will be chosen. In this formalization, *xsplitgate* and *xmergegate* are used for XOR split and merge gateways respectively. An example of a XOR split and merge gateways is `g2` and `g5` in Figure 3.3. For the split gateway `g2`, a decision is made as a result of evaluating associated expressions. To control the divergence, a guard should be defined. We suppose that this guard is part of the gateway itself and not the outgoing sequence flow as the standards in [68] suggests. So it has been defined as an object attribute of a gateway object. The expressions are boolean conditions built as part of the split gateway. For example, in Petri

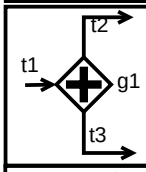
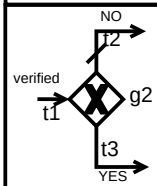
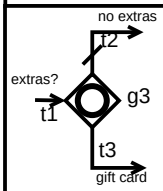
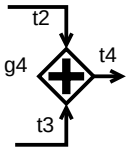
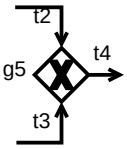
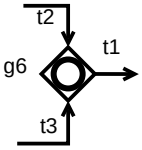
BPMN Gateway	Maude Representation
	**< g 1 : aforkgate | in : t 1 ; out : (t 2,t 3) ; cond : false ; ToBeActive : false ; active : false >**
	**< g 2 : xsplitgate | in : t 1 ; out : (t 2,t 3) ; defaultFlow : t 2 ; error : t 001 ; guard : ((verified == "YES", t 3) . (verified == "NO", t 2)) ; controlValue : noControlValues ; cond : false ; ToBeActive : false ; active : false >**
	**< g 3 : osplitgate | in : t 1 ; out : (t 2,t 3) ; defaultFlow : t 2 ; error : t 002 ; guard : ((extras? == "no extras", t 2) . (extras? == "gift card", t 3)) ; controlValue : noControlValues ; cond : false ; ToBeActive : false ; active : false >**
	**< g 4 : ajoingate | in : (t 2,t 3) ; out : t 4 ; itsSplit : g 1 ; cond : true ; ToBeActive : false ; active : false >**
	**< g 5 : xmergegate | in : (t 2,t 3) ; out : t 4 ; itsSplit : g 2 ; cond : false ; ToBeActive : false ; active : false >**
	**< g 6 : omergegate | in : (t 2,t 3) ; out : t 1 ; itsSplit : g 3 ; cond : true ; ToBeActive : false ; active : false >**

Figure 3.3: Mapping from BPMN Gateways to Maude Representation

net based formalizations for BPs [102, 90, 27], they do not model the conditions and the decision of the flow choice is handled non-deterministically. This allows for ambiguity in interpretations for different executions and the resulting traces.

In order to formally define the guard expressions regardless of the gateway semantics, we provide a Context-Free Grammar (CFG) for the guard expressions. A CFG (sometimes called Backus-Naur Form grammar [16]) is a set of recursive rewriting rules (or productions) used to generate patterns of strings. It was introduced by Chomsky in [17] as a possible way of describing natural languages and then has turned out to be important in describing programming languages (e.g. [8] for ALGOL). In a CFG, the

strings are generated starting by a start symbol, followed by applying one of the production rules with the start symbol on the left hand side [16], replacing the start symbol with the right hand side of the production and the process continues by selecting non-terminal symbols in the string, and replacing them with the right hand side of some corresponding production until all non-terminals have been replaced by terminal symbols.

Definition 3.1.1. (**Guard Expression**) `EG = (N,Σ,S,P)` where

`N = {ConExpr, Cond, X, Y, Z, OP1, OP2}`

`Σ = {∧, ∨, ==, =/=, <=, >=, <, >}`

`S = Cond`

`P = {Cond → ConExpr ∧ Cond | ConExpr,`

`Cond → ConExpr ∨ Cond | ConExpr,`

`ConExpr → X OP1 Y | X OP2 Y | X OP2 Z,`

`OP1 → < | > | <= | >=,`

`OP2 → == | =/= }`

The set of terminals Σ contains the symbols (∧,∨,==, =/=, <=, >=, <) for AND, OR, equal, not equal, less than or equal, greater than or equal, less than, and greater than logical binary operators respectively. The set of nonterminals contains the symbols (`ConExpr, Cond, X, Y, Z, OP1, OP2`) for condition expressions, conditions, expression variable name, numeric variable value, String variable value and comparison operators defined as terminals. The start symbol is `Cond`. The production rules (`P`) describe the possible rewriting steps as follows:

- `Cond` → `ConExpr` ∧ `Cond` | `ConExpr` and `Cond` → `ConExpr` ∨ `Cond` | `ConExpr`: concatenation of two or more conditions with the logical operators ∧ or ∨.
- `ConExpr` → `X OP1 Y` | `X OP2 Y` | `X OP2 Z`: expression structure contains a variable name (`X`), an operator (`OP1` or `OP2`), then a variable value for either numeric (`Y`) or string values (`Z`),
- `OP1` → < | > | <= | >= : the comparison operators for numeric values.
- `OP2` → == | =/=: for comparison operators for string and numeric values.

The proposed EG is used to formalize the guard expressions for the split gateways in our formalization. The guard is linked to a transition which is marking the branch it should follow. Therefore, the guard is a set of a pairs, each one contains the guard expression as the pair first element and the associated transition as the pair second element. We formalize the first part of the pair as the expression of sort Expression (which is following the above CFG), and the second part as the associated transition of sort TransSymbol. The operator (_,_) is used to define this in Maude as indicated below. The operator _._ is used to define the associativity and commutativity properties of the guard expressions. noexp is the guard identity element.

```
op (_,_) : Expression TransSymbol -> Gexp .
op noexp : -> Gexp .
op _._ : Gexp Gexp -> Gexp [ctor assoc comm id: noexp] .
op guard':_ : Gexp -> Attribute .
```

Guard expressions need to be simple and clear to be evaluated in the execution time of a process. The expression contains a variable name and should be assigned a value and it uses one of the boolean operators defined by EG as terminals where Variable, below, represents the variable name and the Nat and String are types of the values of the variable.

```
ops _==_ _=/=_ _<_ _<=_ _>_ _>=_ : Variable Nat -> Expression .
ops _==_ _=/=_ : Variable String -> Expression .
```

This value is supposed to be compared with another value entered by the user to conclude a decision. The second value is entered as an input *control value* and it provides information to the guard to control the flow. An attribute controlValues is defined to capture these values of sort ControlValue, and are associative and commutative with noControlValue as an Identity element.

```
op noControlValue : -> ControlValue [ctor] .
op _.._ : ControlValue ControlValue -> ControlValue
                [ctor assoc comm id: noControlValue] .
op controlValues':_ : ControlValue -> Attribute .
```

An example of an exclusive data-based gateway representation for the gateway g2 in Figure 3.3 where the expression variable name is verified with two possible values defines two expressions (i.e. verified == "YES" and verified == "NO"). Split gateways should have a default flow transition (i.e. an outgoing transition which is chosen in case of no successful guard evaluation) and an error flow transition (i.e. an outgoing, graphically invisible, transition which is used in case of unsuccessful evaluation of all the expressions and the absence of the default flow). For example, in g2 in Figure 3.3, attributes defaultFlow and error. Gateway g2 has two outgoing transitions (or "sequence flows" as will be discussed later in Section 3.1.7). The associated guard condition value is the String value for *verified?*, i.e. "NO" or "YES". The behavioural semantics for the decision gateways is given later in Section 3.4.

A diverging inclusive decision gateway is used to create optional paths within a process flow [68] where the decision is based on conditional expressions defined into the OR split gateway. It should be designed so that at least one path is taken. A default condition could be used to ensure that at least one path is taken. In our formalization, *osplitgate* and *omergegate* are used for OR split and merge gateways respectively. The representation of the OR split and merge gateways is similar to the XOR split and merge gateways representation (e.g. gateways g3 and g6 in Figure 3.3). The OR split gateway (g3) is defined by its incoming and outgoing transitions, guard expressions and associated transitions, and control values, while the OR merge gateway (g6) is defined by its incoming and outgoing transitions and its corresponding split gateway.

3.1.4 Data Objects

A special symbol, d*i*, is used to represent the data object identifiers in this formalization. Recall that data objects can be input, output or data stores as mentioned above. They share common general attributes with other objects like, name and in/out transitions which connect it to the other objects in the process. The attribute isCollection indicates if the data object contains more than one data item (or document) or a single data item, while the attribute linkedObject represents the object which the data object is linked to through the *association flow*. For example, the data object d1 in Figure 3.4 has the name Invoice.

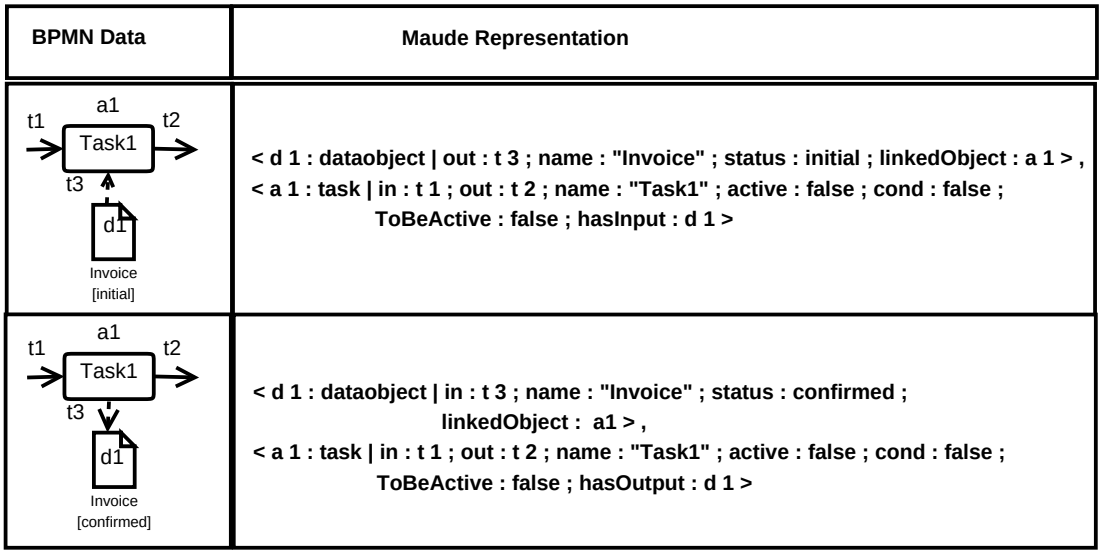


Figure 3.4: Mapping from BPMN Data Objects to Maude Representation

A data object can be assigned a status which can be changed during the process execution. The possible set of statuses for a data object are defined in the set $\texttt{DO}_{state}$, where they depend on the context of the process and defined by the modeller. Attribute `status` represent the data object status in our model. The example data object `Invoice` in Figure 3.4 has two states; `initial`, `confirmed`. The states of data objects are defined syntactically as operators in the specifications and then can be assigned to the attribute `status`, as for `d1`.

```
ops initial confirmed : -> DOstate .
op status':_ : DOstate -> Attribute .
```

The object linked to the data object has an attribute reference to the data object using the attribute `hasOutput` if the data object is produced by that activity (output; as in the activity `a2` in Figure 3.4) or the attribute `hasInput` if it is consumed by that activity (input; as in the activity `a1` in Figure 3.4).

3.1.5 Swimlanes

BPMN swimlanes represent participants in the business process. Generally, lanes are often used for internal roles (e.g. manager), systems (e.g. an enterprise application), or an internal department (e.g. shipping, finance), while pools are often used for external entity (e.g. company, third party, government). We model swimlanes as attributes `pool` and

`lane` attached to each object in the process representing to which participant they belong. For example in Figure 3.2, the object `a1` in the last row has the attribute `pool:"p1"`.

3.1.6 Artifacts

Artifacts can be groups or text annotations [68]. The groups represent the boarders for a set of related objects in the BP (e.g. for classification and documentation purposes), without affecting the execution of the process. As a result, the group are represented in our formalization as a String attribute referring to the group name (i.e. `op group':_ : String -> Attribute`).

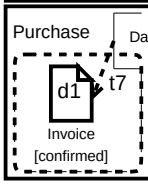
BPMN Artifacts	Maude Representation
	**< tann 1 : textAnnotation | out : t 7 ; name : "Data Object" ; linkedObject : d 1 ; group : "Purchase" >**

Figure 3.5: Mapping from BPMN Artifacts to Maude Representation

The text annotation is usually linked to a certain object in the process and hence, more specific information need to be modelled to describe a text annotation. A text annotation is represented as an object with `tann`*i* as an object identifier, `textAnnotation` as its sort, and contains the attributes connecting it to a certain object (i.e. `sourceObject`) and the text it holds (i.e. `name`) beside the association flow (i.e. `out`). For example, the text annotation `tann1` and the attribute `group` in Figure 3.5.

```
op tann_ : Nat -> Oid .
op textAnnotation : -> Cid .
```

3.1.7 Connecting Objects

Connecting objects are represented by unique *"flow transitions"* in the proposed formalization, where each one forms a link between two objects in the BPMN process.

```
ops normalflow uncontrolledflow conditionalflow
    defaultflow exceptionflow : -> SequenceFlow [ctor] .
```

```
op messageflow : -> MessageFlow [ctor] .
op association dataAssociation : -> Association [ctor] .
```

A `sequenceflow` specifies the order of flow elements in a process. Assuming that the attributes `sourceObject` and `targetObject` indicate the source and target objects for a transition object, the following are the sequence flows types according to [68]. Following the items in Figure 3.6, (1) the normal flow originates from start event and continues through activities on alternative and parallel paths until an end event is reached, e.g. `t1`. It forms paths of sequence flow that do not start from an intermediate event attached to the boundary of an activity. (2) uncontrolled flow proceeds without dependencies or conditional expressions (e.g. a flow between two activities that do not have a conditional indicator (mini-diamond) or an intervening gateway)), e.g. `t2`, (3) conditional flow proceeds from one flow object to another, via a sequence flow link, but is subject to either conditions or dependencies from other flow, e.g. `t3` in the figure, (4) default flow proceeds from a decision based gateway and used only if all the other outgoing conditional flow are not true at runtime, e.g. `t4`, (5) exception flow originates from an intermediate event attached to the boundary of an activity. The process does not traverse this path unless the activity is interrupted by the triggering of a boundary exception, e.g. `t5`, (6) message flow shows the flow of messages between two participants, and `messageflow` represents its object sort as in the object `t6`, and (7) association links information and artifacts with flow objects. An `association` is an object connects the text annotation to other objects which may contain descriptive information about them, e.g. `t7` in Figure 3.6.

In Section 3.4, we are not using the transitions (connecting objects) as a whole separated objects in the formalization. They are still referred to into the objects using attributes `in` and `out`, however, the process representation in Maude will ignore their full object description for sake of simplicity. For the conditional flow following the decision gateways, we propose a novel approach to consider the conditional expressions as part of the gateway themselves, and to be evaluated into them as well as the domain specific rules defining specific flow condition, as detailed in Section 3.4.

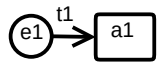
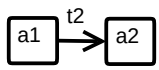
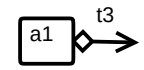
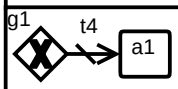
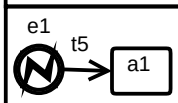
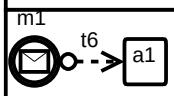
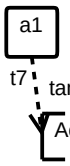
Connecting Flow	Maude Representation
	**< t 1 : sequenceFlow | flowType : normal ; sourceObject : e 1 ; targetObject : a 1 >**
	**< t 2 : sequenceFlow | flowType : uncontrolled ; sourceObject : a 1 ; targetObject : a 2 >**
	**< t 3 : sequenceFlow | flowType : conditional ; sourceObject : a 1 ; targetObject : a 2 >**
	**< t 4 : sequenceFlow | flowType : default ; sourceObject : g 1 ; targetObject : a 1 >**
	**< t 5 : sequenceFlow | flowType : exception ; sourceObject : e 1 ; targetObject : a 1 >**
	**< t 6 : messageFlow | sourceObject : m 1 ; targetObject : a 1 >**
	**< t 7 : association | sourceObject : a 1 ; targetObject : tann 1 ; name : "Activity" >**

Figure 3.6: Mapping from BPMN Connecting Flow to Maude Representation

3.2 Introducing Example

In order to give a better explanation of the proposed formalization, we introduce an example in Figure 3.7 which will be used to demonstrate the proposed formalized BPMN behaviour. It represents a process model called *Release Baseline*, a subprocess of the Configuration Management (CM) process, where a configuration item (CI) is an entity designated for one or more related work products such as tangible assets (e.g. hardware) and intangible assets (e.g. software, OS) [22]. A collection of CIs that are used in a project or a company may be baselined (i.e. considered a baseline document) whenever they are sufficiently stable, enabling a more strict control for changing them.

In the process *Release Baseline*, access requests are check for being an authorized access or not using the document *Authorization List*. If the access authorization is granted, a change request (CR) is chosen and the authorized changes are passed through to the (`open CR`) activity. While a specific CR is open, the related CI is retrieved, changed, and documented before the CR is closed. If there is more than one CR waiting, the same

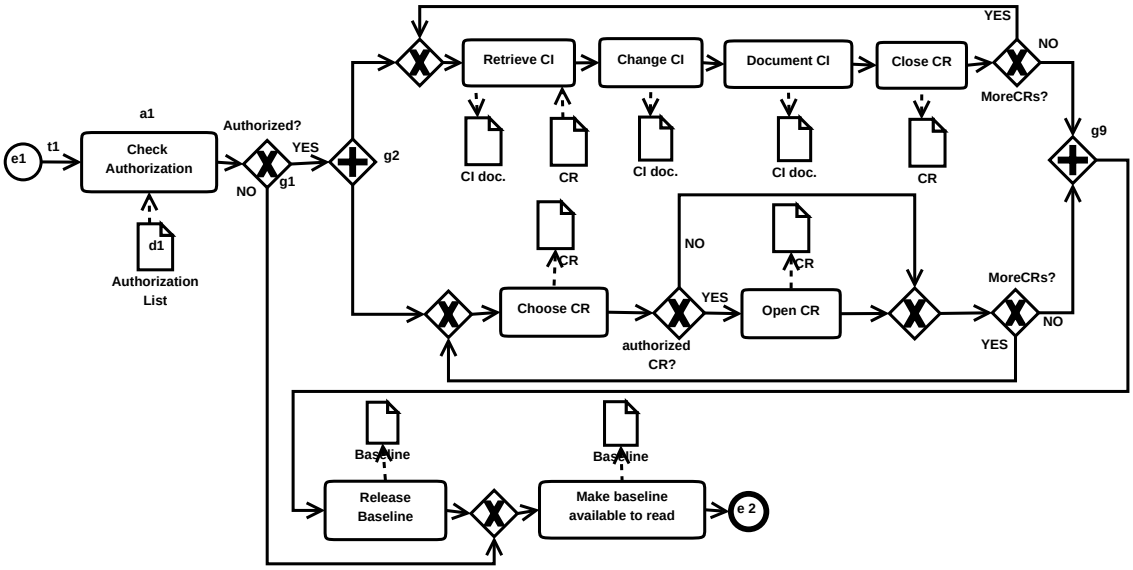


Figure 3.7: Release Baseline Process - BPMN representation

procedure is repeated until no more CRs are left in the process. After that the baseline is released with all the changes. This is followed by making the released baseline available for stakeholders (in case of updating the `CI doc`) or the latest approved baseline (in case of declining the authorized access). Notice the block gateway structure for gateways: `g1` and `g10`, `g2` and `g9`, `g3` and `g4`, `g5` and `g8`, `g6` and `g7`. Structured loop gateway blocks decide either to forward the process to later actions or to go back to the merge gateways (i.e. `g3` and `g5`). Figure 3.8 shows the Maude representation for the process. In the following section, A well-formed BPMN model is defined and in Section 3.4, we introduce the BPMN process semantics.

3.3 Well-Formed BPMN Processes

In order to obtain a structured BPMN processes, the following requirements, which are extracted from the BPMN standard document [68] are introduced, where the symbol $|X|$ represents the *number* of elements in X and the functions: *ObjId*, *ObjCid*, *in*, *out*, *sourceObject*, *targetObject*, *eventType*, and *pool* for: object identifier, class identifier, input transitions, output transitions, source object, target object, type of an event, and the pool name are operators defined as follows.

Function ($ObjId : Object \rightarrow Oid$) is defined to return the object identifier for an ob-

```
<< < e 1 : startEvent | eventType : start ; in : notrans ; out : t 1 ; process : true ; cond : false ; ToBeActive : false ; active : false > ,
 < a 1 : task | name : "Check Authorization" ; in : t 1 ; out : t 2 ; hasInput : d 1 ; cond : false ; ToBeActive : false ; active : false > ,
 < d 1 : dataobject | name : "Authorization List" ; out : t 444 ; linkedObject : a 1 ; status : none > ,
 < g 1 : xsplitgate | in : t 2 ; out : (t 3, t 4) ; defaultFlow : t 3 ; guard : ((Authorized? == "NO", t 3) . (Authorized? == "YES", t 4)) ;
                      cond : false ; ToBeActive : false ; controlValues : noControlValue ; error : t 100 ; active : false > ,
   < g 2 : aforkgate | in : t 4 ; out : (t 5, t 6) ; cond : false ; ToBeActive : false ; active : false > ,
     < g 3 : xmergegate | in : (t 5, t 12) ; out : t 7 ; cond : false ; ToBeActive : false ; active : false > ,
       < a 2 : task | name : "Retrieve CI" ; in : t 7 ; out : t 8 ; hasInput : d 2 ; hasInput : d 3 ; cond : true ; ToBeActive : false ;
                      active : false > ,
       < a 3 : task | name : "Change CI" ; in : t 8 ; out : t 9 ; cond : false ; ToBeActive : false ; active : false > ,
       < a 4 : task | name : "Document CI" ; in : t 9 ; out : t 10 ; hasOutput : d 3 ; cond : false ; ToBeActive : false ; active : false >,
       < a 5 : task | name : "Close CR" ; in : t 10 ; out : t 11 ; hasOutput : d 2 ; cond : false ; ToBeActive : false ; active : false > ,
       < d 2 : dataobject | name : "CR" ; out : t 445 ; linkedObject : (a 2, a 7) ; status : initial > ,
       < d 3 : dataobject | name : "CI doc" ; out : t 446 ; linkedObject : (a 2, a 3, a4) ; status : initial > ,
     < g 4 : xsplitgate | in : t 11; out : (t 12,t 13); defaultFlow : t 13; guard : ((MoreCRs1?=="YES",t 12).(MoreCRs1?=="NO",t 13)) ;
                          cond : false ; ToBeActive : false ; controlValues : noControlValue ; error : t 100 ; active : false > ,
     < g 5 : xmergegate | in : (t 6, t 20) ; out : t 14 ; cond : false ; ToBeActive : false ; active : false > ,
     < a 6 : task | name : "Choose CR" ; in : t 14 ; out : t 15 ; hasOutput : d 2 ; cond : false ; ToBeActive : false ; active : false > ,
     < g 6 : xsplitgate | in : t15; out : (t16,t17); defaultFlow:t16; guard : ((AuthorizedCR?=="YES",t17).(AuthorizedCR?=="NO",t16));
                          cond : false ; ToBeActive : false ; controlValues : noControlValue ; error : t 100 ; active : false > ,
       < a 7 : task | name : "Open CR" ; in : t 17 ; out : t 18 ; hasOutput : d 2 ; cond : true ; ToBeActive : false ; active : false > ,
     < g 7 : xmergegate | in : (t 16, t 18) ; out : t 19 ; cond : false ; ToBeActive : false ; active : false > ,
     < g 8 : xsplitgate | in : t19 ; out : (t20, t21) ; defaultFlow : t21 ; guard : ((MoreCRs2?=="YES",t20).(MoreCRs2?=="NO",t21));
                          cond : false ; ToBeActive : false ; controlValues : noControlValue ; error : t 100 ; active : false > ,
   < g 9 : ajoingate | in : (t 13, t 21) ; out : t 22 ; itsSplit : g 2 ; cond : true ; ToBeActive : false ; active : false > ,
   < a 8 : task | name : "Release Baseline" ; in : t 22 ; out : t 23 ; hasOutput : d 4 ; cond : false ; ToBeActive : false ; active : false > ,
 < g 10 : xmergegate | in : (t 3, t 23) ; out : t 24 ; cond : false ; ToBeActive : false ; active : false > ,
 < a 9 : task | name : "Make Baseline Available to Read" ; in : t 24 ; out : t 25 ; hasOutput : d 4 ; cond : false ; ToBeActive : false ;
                active : false > ,
 < d 4  : dataobject | name : "Baseline" ; linkedObject : (a 8, a 9) ; out : t 447 ; cond : false ; ToBeActive : false ; active : false > ,
< e 2 : endEvent | eventType : end ; in : t 25 ; out : notrans ; cond : false ; ToBeActive : false ; active : false > >> .
```

Figure 3.8: Release Baseline Process - Maude representation

ject, where *Oid* is the set of objects' identifiers in the process model. For example, $ObjId(< a_1 : task | name : "CheckAuthorization"; in : t_1; out : t_2; active : true >) = a_1$.

Function ($ObjCid : Object \rightarrow Cid$) is defined to return the object class identifier, where *Cid* is the set of objects' class identifiers (types) in the model. For example, $ObjCid(< a_1 : task | name : "CheckAuthorization"; in : t_1; out : t_2; active : true >) = task$.

Functions *in* and *out* are defined as ($in : Object \rightarrow T$) and ($out : Object \rightarrow T$), where T is the set of connecting flows defined in Definition 2.1.1 and T is the transition flows in the process O (refer to Definition 2.1.1).

Functions *sourceObject* and *targetObject* are defined as ($sourceObject : Object \rightarrow Oid$) and ($targetObject : Object \rightarrow Oid$), where *Oid* is the object identifier.

Function *pool* is defined as ($pool : Object \rightarrow String$), where it returns the *String* name of the pool which an input object belongs to.

Function *eventType* is defined as ($eventType : Object \rightarrow TypeofEvent$), where it returns the value of the `eventType` attribute in an event object.

Definition 3.3.1. (**Well-Structured Process**) A Well-Structured BPMN process `S-BPMN` is a BPMN process $O = (OS, T)$[4] where o, o_1, o_2, o_3 are symbols for arbitrary objects in OS such that the following hold.

1. Start and exception events have no incoming flows and have one outgoing flow; i.e. $\forall o \in OS(ObjCid(o) \in \{\texttt{startEvent}, \texttt{exception}\} \rightarrow (|in(o)| = 0 \wedge |out(o)| = 1))$ and

2. An end event has no outgoing flows and has one incoming flow; i.e. $\forall o \in OS(ObjCid(o) = \texttt{endEvent} \rightarrow (|in(o)| = 1 \wedge |out(o)| = 0))$ and

3. A process that has an end event, must have a start event; i.e. $\forall o_1 \in OS(ObjCid(o_1) = \texttt{endEvent} \rightarrow \exists o_2 \in OS(ObjCid(o_1) = \texttt{startEvent}))$ and

4. Message flows must connect two separate pools. They must not connect two objects within the same pool; i.e. $\forall o_1 \in E_I(eventType(o_1) = \texttt{message} \rightarrow \exists o_2, o_3 \in OS(sourceObject(o_1) = ObjId(o_2) \wedge targetObject(o_1) = ObjId(o_3) \wedge pool(o_2) \neq pool(o_3)))$ and

5. An artifact must not be a target/source for a sequence flow or a message flow; i.e. $\forall o_1 \in OS(ObjCid(o_1) \in \{\texttt{txtAnnotation}, \texttt{group}\} \rightarrow \neg\exists o_2 \in T_S \cup T_M(sourceObject(o_2) = ObjId(o_1) \vee targetObject(o_2) = ObjId(o_1)))$.

Based on the definition above, a well-structured BPMN process can have more than one start/end event, a split gateway without a corresponding merge gateway, and a gateway can have multiple incoming and outgoing transitions at the same time, which considered ambiguity representation. In order to avoid that in our formal representation of the BPMN processes, we introduce the well-formed BPMN. Inspired by the work in [35, 70, 107], we define the well-formed core subset of the BPMN elements. A BPMN process is said to be well-formed if its elements satisfy the properties in Definition 3.3.1. For these properties, an equationally-defined boolean predicate characterising them is introduced afterwards. The operator `itsSplit` takes a merge gateway object identifier and returns its corresponding split gateway object identifier, ($itsSplit : Object \rightarrow Oid$). The well-formed BPMN process definition includes references to the functions in Maude

[4] Definition 2.1.1

implementing the conditions. The functions are defined and explained in the rest of the section. For example, points 1 ad 2 are implemented using function (`wfstartend`).

Definition 3.3.2. (**Well-formed BPMN model**): A Well-Formed BPMN process (`W-BPMN`) is a well-structured BPMN process (`S-BPMN`) such that the following hold.

1. A process should have one start event; i.e. $\exists o_1 \in OS(ObjCid(o_1) = \texttt{startEvent} \rightarrow \neg\exists o_2 \in OS(ObjCid(o_2) = \texttt{startEvent} \wedge o_1 \neq o_2))$ (function `wfstartend`) and

2. A process should have one end event; i.e. $\exists o_1 \in OS(ObjCid(o_1) = \texttt{endEvent} \rightarrow \neg\exists o_2 \in OS(ObjCid(o_2) = \texttt{endEvent} \wedge o_1 \neq o_2))$ (function `wfstartend`) and

3. Activities and non-exception intermediate events should have one input and one output transition flows; i.e. $\forall o \in A \cup E_I \setminus \texttt{exception}(\exists t_1, t_2 \in T(in(o) = t_1 \wedge out(o) = t_2 \wedge t_1 \neq t_2 \neq \texttt{notrans})$ (functions `wfException`,`wfActivities`) and

4. Fork and decision gateways should have one input transition flow and at least two output transition flows; i.e. $\forall o \in OS(ObjCid(o) \in \{\texttt{ANDfork}, \texttt{XORsplit}, \texttt{ORsplit}\} \rightarrow |in(o)| = 1 \wedge |out(o)| > 1)$ (function `wfGates`) and

5. Join and merge gateways should have one output transition flow and at least two transition flows as inputs; i.e. $\forall o \in OS(ObjCid(o) \in \{\texttt{ANDjoin}, \texttt{XORmerge}, \texttt{ORmerge}\} \rightarrow |in(o)| > 1 \wedge |out(o)| = 1)$ (function `wfGates`) and

6. Block structure. Except for exception events, each split gateway has a corresponding merge gateway from the same type, forming a *block* in the model. Exception objects attached to an activity boundary split the flow and then the flow is merged with the normal flow using an XOR merge gateway, enforcing gateways block structure (function `wfGates`). The following statements must hold:

   (a) $\forall o_1 \in OS(ObjCid(o_1) = \texttt{aforkgate} \rightarrow \exists o_2 \in OS(ObjCid(o_2) = \texttt{ajoingate} \wedge \texttt{itsSplit}(o_2) = ObjId(o_1)))$

   (b) $\forall o_1 \in OS(ObjCid(o_1) = \texttt{xsplitgate} \rightarrow \exists o_2 \in OS(ObjCid(o_2) = \texttt{xmergegate} \wedge \texttt{itsSplit}(o_2) = ObjId(o_1)))$

   (c) $\forall o_1 \in OS(ObjCid(o_1) = \texttt{osplitgate} \rightarrow \exists o_2 \in OS(ObjCid(o_2) = \texttt{omergegate} \wedge \texttt{itsSplit}(o_2) = ObjId(o_1)))$

(d) $\forall o_1 \in OS(\texttt{eventType}(o_1) = \texttt{exception} \rightarrow \exists o_2 \in OS(ObjCid(o_2) = \texttt{xmergegate} \wedge \texttt{itsSplit}(o_2) = ObjId(o_1)))$,

7. Every object should be on a *complete path* from the start (or an exception event) to the end event; i.e. $\forall o_1 \in OS(\exists o_2, o_3 \in OS\ (((ObjCid(o_2) = \texttt{startEvent} \vee eventType(o_2) = \texttt{exception}) \wedge ObjCid(o_3) = \texttt{endEvent}) \rightarrow (o_2 \in preds(o_1, OS) \wedge o_3 \in succs(o_1, OS)))$, (function `wfpath`).

In point (6) in Definition 3.3.2, gateways are required to be designed, in our formalization, as a *block* in the model, i.e. each split gateway should have an accompanying merge gateway of the same type. A block has only one entrance point and one exit point, e.g. the split gateway input flow and the merge gateway output flow respectively in the case of acyclic models and the other way around in case of feedback cases (structured loops). Notice that the definitions above does not exclude the loop structure from being a well-formed model (i.e. a XOR merge gateway followed, at some point after it, by a XOR split decision gateway). In point (7), we use functions *preds* and *succs* to retrieve the set of predecessors and successors for a certain object in the process respectively.

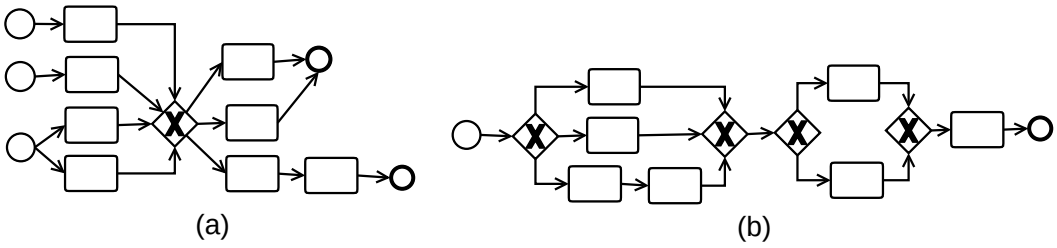


Figure 3.9: (a) a `S-BPMN` model (b) a `W-BPMN` model.

In Figure 3.9, an example of the differences between a well-structured and a well-formed model is graphically illustrated. The model in (a) has a gateway with more than one input and output transitions and more than one start and end events at the same time, while the model in (b) satisfies the well-formedness requirements above.

In order to describe the BPMN model with respect to the dependency relationships among its objects, we define the notions of path and complete path in a well-formed BPMN model. The elements of the process O[5] are represented by objects o_i where $1 \leq$

[5] O is defined in Definition 2.1.1 as the pair (OS, T) of the set of objects OS and the set of transition flows T

$i < n$, and $o_i \in OS$. Objects $o_s \in OS$ and $o_e \in OS$ are a start and end events respectively.

Definition 3.3.3. (**Path**): A path P from object o_1 to o_n is a finite sequence $(o_1, o_2, \ldots, o_n)$ of objects such that $out(o_i) \cap in(o_{i+1}) \neq \emptyset$ for $1 \leq i < n$.

Definition 3.3.4. (**Complete Path**): A path is complete (*cP*) if it starts with a start event o_s and ends with an end event o_e.

In order to automate the check of the well-formedness conditions, we introduce equationally-defined predicate (`wfs`) for well-formed set of BPMN elements. It checks whether the requirements are satisfied for each set of elements. It takes as inputs an object and the process (set of objects) it belongs to and retrieves a boolean value; *true* if the conditions satisfied, and *false* if at least one of the conditions is not satisfied.

```
op wfs : Object ObjectSet -> Bool .
var O : Object .
var A : ObjectSet .
ceq wfs (O,A) = true
 if wfstartendTF((O,A),noobject) /\ wfExceptionTF((O,A),noobject) /\
    wfActivityTF((O,A),noobject) /\ wfGatesTF((O,A),noobject) /\
    wfpathTF(O,(O,A),noobject) .
eq wfs(O,A) = false [owise] .
```

A comparison is conducted between the set[6] which contains the well-formed objects of the same type and the set of the objects of that particular type in the process. If these two sets are identical, the condition is satisfied. Otherwise (`[owise]` function above), it returns *false*. The function condition above is broken down into smaller detailed (more intuitive) sub-conditions. That means, in case of `wfs` returned *false*, the modeller can still know which object type exactly has the problem, by tracing the results of the sub-conditions in the function.

The condition contains a set of helpful functions, which we are presenting the definition of one of them (i.e. `wfstartendTF`) as an example and the rest are included into Appendix B. The function (`wfstartendTF`) takes two object sets and returns *true*

[6]We use the operator `noobject` as the identity element for the `objectSet`

if they are equal, and *false* if they are not. The functions defined below are: function `startendCollector` to collect the start and end events, function `wfstartend` to collect only the well-formed start and end events, while function `wfstartendTF` to decide if the two output set are equal (i.e. the start and end events in a process are well-formed). In function `wfstartendTF`, if the two input sets are not equal, that means that not all the start/end events are well-formed. There might be a start event with more than one outgoing transition, or an end event with outgoing transitions.

```
op startendCollector : ObjectSet ObjectSet -> ObjectSet .
eq startendCollector((< E1 : startEvent | AS1 >, A), B)
 = startendCollector(A, (< E1 : startEvent | AS1 >, B)) .
eq startendCollector((A,< E1 : endEvent | AS1 >),B)
 = startendCollector(A,(< E1 : endEvent | AS1 >,B)) .
eq startendCollector(A,B) = B [owise] .

op wfstartend : ObjectSet ObjectSet -> ObjectSet .
eq wfstartend((<E1:startEvent|in:notrans;out:tN1;AS1>,A),B)
 = wfstartend(A,(<E1:startEvent|in:notrans;out:tN1;AS1>,B)) .
eq wfstartend((<E1:endEvent|in:tN1;out:notrans;AS1>,A),B)
 = wfstartend(A,(<E1:endEvent|in:tN1;out:notrans;AS1>,B)) .
eq wfstartend(A,B) = B [owise] .

op wfstartendTF : ObjectSet ObjectSet -> Bool .
ceq wfstartendTF(A,noobject) = true
 if startendCollector(A,noobject) = wfstartend(A,noobject) .
eq wfstartendTF(A,B) = false [owise] .
```

The same idea is applied to the exception events (i.e. function `wfExceptionTF`), the activities (i.e. function `wfActivityTF`), the gateways (i.e. function `wfGatesTF`), and the complete paths for all the objects in a process (i.e. function `wfpathTF`). The definition of the functions are included into Appendix B.

An example of a well-formed BPMN model tested using these designed Maude functions is presented in Figure 3.10. Example1 represents a well-formed model while Ex-

ample2 is not. Example2 model has a dangling activity a3 which has no output transition. Moreover, activity a3 has two incoming transitions which violates the second condition in Definition 3.3.2. By running the two models through function wfs, the results shows that Example1 is well-formed and Example2 is not. In Example2, the condition wfActivityTF evaluates to *false* because activity a3 has two incoming transitions and has no outgoing transition.

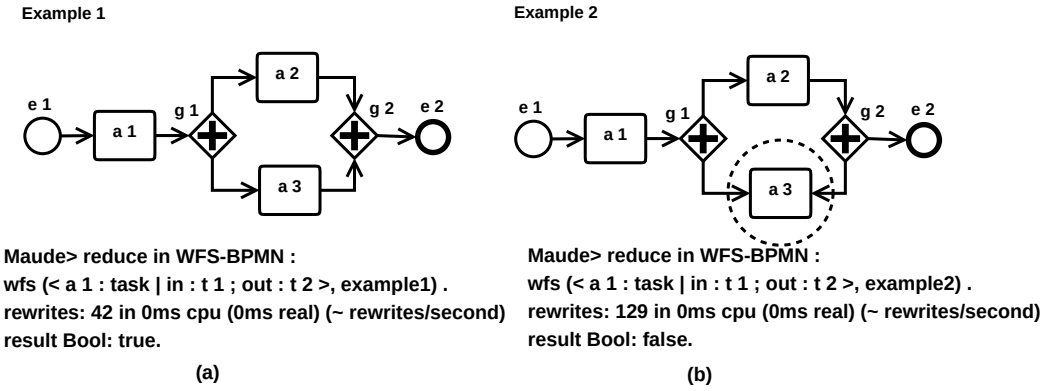


Figure 3.10: *wfs* results for (a) W-BPMN and (b) not W-BPMN models.

If the process satisfies the above well-formedness requirements, we can consider it of type well-formed process instead of a set of objects (ObjectSet). On satisfying the well-formedness requirements, a process is a WFprocess (i.e. a subsort of the main sort ObjectSet). Following Maude's conditional membership [58], an object can change its sort during execution. The membership can be conditional like the one used below.

```
subsort  WFprocess < ObjectSetConf .
op <<_>> : ObjectSet -> ObjectSetConf [ctor].
cmb << O, A >> : WFprocess if wfs(O,A) .
```

The code above specifies that a set of objects is well-formed (of sort WFSprocess) only if it satisfies the well-formedness conditions as described in Definition 3.3.2 and coded in wfs function.

3.4 BPMN Formal Semantics Specifications

Following the introduced BPMN syntax for well-formed models in Section 3.1, this part of the chapter focuses on the behavioural aspects of the BPMN elements and how the models can be executed/simulated. Two categories of semantics specification rules are defined. The first category contains the general rules for common pattern of behaviour in a BPMN model (in Section 3.4.2), while the second category contains more domain specific rules (in Section 3.4.7) for the introduced example in Section 3.2. In addition, an evaluation mechanism is proposed for guard expressions in decision-based gateways (i.e. XOR and OR split gateways) based on the guard CFG introduced in Section 3.1.3. This is followed by a detailed formalization of the behaviour attached to the exception events, message events, and data objects. In particular, the data objects formalization (in Section 3.4.6) as a main resource in the business process identifies the involved documents, reports and forms which mark and document different stages in the business process. In Appendix B, Section C.1 presents some helpful functions defined to validate the BPMN model formalization and introducing some operations on the BPMN processes.

3.4.1 Process State Model

In our approach, the execution of well-formed BPMN processes is based on the notion of activation. A well-formed BPMN process is *active* if one or more of its objects are active. A well-formed BPMN process is *inactive* if all its objects are inactive. In the BPMN process, each flow object has its own state which can be *active*, if the object is being executed, *inactive*, if the objects is not in execution, or *ready2bActive*, if the object is waiting for a condition to be fulfilled. The boolean attribute `active` is used to indicate whether an object is *active* or *inactive* and the boolean attribute `ToBeActive` to identify the *ready2bActive* state. Therefore, we have $F_{state} = \{active, inactive, ready2bActive\}$ as the set of the flow objects states. Data object states are user-defined values entered at design time. Therefore, we have $D_{state} = \{s_{do} | s_{do}$ is a user-defined value of data objects statuses in the model$\}$ as the set of the data objects states in a process model.

Given an object o, we define the function ($s : Object \rightarrow O_{state}$) to take a flow (or data) object and returns the object state (i.e. s_{fo} or s_{do}) as *active* if the object is active,

ready2bActive if the object is ready to be activated, *inactive* if the object is inactive or data object status (i.e. s_{do}). For example,

$s(< a_1 : task \mid name : "CheckAuthorization"; in : t_1; out : t_2; active : true >) = active$

$s(< d_1 : dataobject \mid name : "AuthorizationList"; linkedObject : a_1; status : declined >) = declined$,

where $declined \in D_{state}$.

The union of both sets forms the set of object states in a BPMN model (O_{state}), i.e. $O_{state} = F_{state} \uplus D_{state}$. Therefore, $\forall o \in FO(\exists s_{fo} \in F_{state}(s(o) = s_{fo})) \wedge \forall o \in DO(\exists s_{do} \in D_{state}(s(o) = s_{do}))$.

In the following we define the process state, state space and some special process states.

Definition 3.4.1. (**Process State**): For a BPMN process $O = (OS, T)$[7], where the set of objects $OS = FO \uplus DO$ contains the set of flow objects and data objects, the process state S_{OS} for the process O is defined as the set of pairs with the first element as the object o's identifier and the second element as the object o's state; i.e.

$S_{OS} = \{(o_{id}, s_{fo}) \mid o_{id} = ObjId(o), o \in FO, s_{fo} \in F_{state}\} \uplus \{(o_{id}, s_{do}) \mid o_{id} = ObjId(o), o \in DO, s_{do} \in D_{state}\}$.

Definition 3.4.2. (**Process State Space**): For a BPMN process $O = (OS, T)$ defined by the rewrite theory $\mathcal{R} = (\Sigma, Eq \cup U, \varphi, R)$, the set $P(O)$ of possible process states (i.e. state space) of O is defined as the set of all possible states resulting from applying the rewrite rules R to the process, i.e. $\{S_{OS'} | S_{OS} \xrightarrow{*}_{\mathcal{R}} S_{OS'}\}$.

Definition 3.4.3. (**Special Process States**): For a `W-BPMN` process $O = (OS, T)$, the following are special process states where $o \in OS$ is an arbitrary object in the process.

1. **Active State**: at lease one object is active or ready to be active; i.e.
   $\exists (o_{id}, s_{fo}) \in S_{OS}(s_{fo} \in \{active, ready2bActive\})$,

2. **Inactive State**: all objects are inactive; i.e. $\forall (o_{id}, s_{fo}) \in S_{OS}(s_{fo} = inactive)$,

3. **Start State** (S_s): the only active object is the start event; i.e.
   $(\exists (o_{id}, s_{fo}) \in S_{OS}(o_{id} = ObjId(o) \wedge ObjCid(o) = \texttt{startEvent} \wedge s_{fo} = active) \wedge$
   $\neg \exists (o'_{id}, s'_{fo}) \in S_{OS}(o'_{id} = ObjId(o') \wedge s'_{fo} \in \{active, ready2bActive\}) \wedge o \neq o') \Rightarrow$
   $S_{OS} = S_s$, and

[7] Definition 2.1.1 of a BPMN process in Chapter 2.

4. **End State** (S_e): the only active object is the end event; i.e.
   $(\exists(o_{id}, s_{fo}) \in S_{OS}(o_{id} = ObjId(o) \wedge ObjCid(o) = \texttt{endEvent} \wedge s_{fo} = active) \wedge$
   $\neg\exists(o'_{id}, s'_{fo}) \in S_{OS}(o'_{id} = ObjId(o') \wedge s'_{fo} \in \{active, ready2bActive\}) \wedge o \neq o') \Rightarrow$
   $S_{OS} = S_e$.

For simplicity, we are going to use S and S' instead of S_{OS} and $S_{OS'}$ in the remaining of the thesis. Moreover, we can refer to the set of process states (i.e. state space) $P(O) = \{S_{OS1}, S_{OS2}, \ldots, S_{OSn}\}$ as $P(O) = \{S_1, S_2, \ldots, S_n\}$. The *Release Baseline* process model in Figure 3.8 is in its inactive state, as all the objects are inactive. The next section introduces the general behavioural semantics for BPMN elements following the document [68]. This will cover the sequential, parallel, exclusive and inclusive decision-based behaviours. The semantics is translated using Maude.

3.4.2 General Behaviour Rules

Generally, if a rewrite rule $r \in R$ is applied to a process, it changes its state from state S to another process state S'. In Maude, the rules are applied if there is a matching found in a term with the left-hand side pattern of the rule. That is, a state is transformed into another state through the application of rewrite rules which model the behavioural semantics of the process. A state S' is reachable from state S if and only if S' can be obtained by applying zero or more rewrite rules to state S; i.e. $S \xrightarrow{*} S'$[8].

Definition 3.4.4. (**Execution Step**): An execution step *es* is a triple (S, r, S') of an input state S, an applicable rewrite rule r, and the resulting state S'. It can be represented as $(S \xrightarrow{r} S')$ (i.e. $\exists r \in R(S \xrightarrow{r} S')$).

We write $s_{fo} \xrightarrow{r} s'_{fo}$ or $s_{do} \xrightarrow{r} s'_{do}$ where s_{fo} and s'_{do} are the states for an object o affected by the rewrite rule r, and S and S' are the corresponding process states respectively. A change in one object state can change the process state, while a change in a process state may indicate that one or more objects have changed their states (i.e. the general behaviour rewrite rules, as will be explained below, are changing the state of more than one object at the same time, while some of the domain specific rules changes the state of

[8] We use the symbol $\xrightarrow{*}$ to denote a sequence of zero or more rewrite steps and the symbol $\xrightarrow{+}$ to denote a sequence of one or more rewrite steps.

a data object or an object at a time). Here we define the function ($source : es \rightarrow P(O)$), and the function ($target : es \rightarrow P(O)$), to return the source and target states in an input execution step where $P(O)$ is the set of process states for a process O. For example, $source(S,r,S') = S$ and $target(S,r,S') = S'$. The set of execution steps for a BP model forms the execution path for the process. So that, we define the execution path as the finite sequence of execution steps; where the source of an execution step equals the target of its successor.

Definition 3.4.5. (**Execution Path**): An execution path $\mathcal{EP}_O$ of a process O is a finite sequence $(es_1, es_2, \ldots, es_n)$ of execution steps, such that $target(es_i) = source(es_{i+1})$ for all $1 \leq i < n$.

We define the operator ($Seq2Set : Sequence \rightarrow Set$) to take a sequence of elements and return the equivalent set containing the same elements. The operator is used to generate the set $\mathcal{EP}$ of the execution path $\mathcal{EP}_O$ (i.e. $Seq2Set(\mathcal{EP}_O) = \mathcal{EP}$ where if $\mathcal{EP}_O = (es_1, es_2, \ldots, es_n)$, then $\mathcal{EP} = \{es_1, es_2, \ldots, es_n\}$). Let O_{Paths} denotes the set of execution paths for a process O.

Definition 3.4.6. (**Complete Execution Path**): An execution path $\mathcal{EP}_O$ of a process O is complete if it contains an execution step which is sourced from a start state S_s and another execution step which target is an end state S_e.

Using the operator *Seq2Set* we define the set $c\mathcal{EP}$ of all execution steps of a complete execution path $c\mathcal{EP}_O$ of O as $c\mathcal{EP} = Seq2Set(c\mathcal{EP}_O)$.

In the following subsections, a Maude formalization for BPMN 2.0 semantics is introduced and detailed. The following are some of the variables which are used in the rules.

```
vars X Y Z : Oid .              vars K L M : Cid .
vars T1 T2 T3 : TransSymbol .   vars A B C D : ObjectSet .
vars O O1 O2 : Object .         vars N1 N2 N3 : Nat .
vars S1 S2 S3 PN1 : String .    vars G1 G2 : GateSymbol .
var GCid1 GCid2 : GateCid .     vars E1 : EventSymbol .
vars D1 D2 : DataSymbol .       var DT : DataType .
vars GExp1 : Gexp .             var CVcol : CVcollection .
```

```
vars P Q : Bool .              vars V1 V2 : Variable .
vars CVs1 CVs2 CVs3 : ControlValue .
vars AS1 AS2 AS3 : AttributeSet .
```

Initiating and Terminating the Process

According to the well-formed BPMN model requirements in Section 3.3, a well-formed process should have one start object which is supposed to be the first to execute if the process is initiated. The start event is activated according to the rule `InitiateProcess` which assigns the value *true* to the start event `active` attribute if there are no active objects in the process initial state. This transition changes the process state from inactive to active. Rule `InitiateProcess` activates the start event of the process if the function `isActive` retrieve *false*.

```
op isActive : ObjectSet -> Bool .
eq isActive(< X : K | active : true ; AS1 >, A) = true .
eq isActive(A) = false [owise] .
```

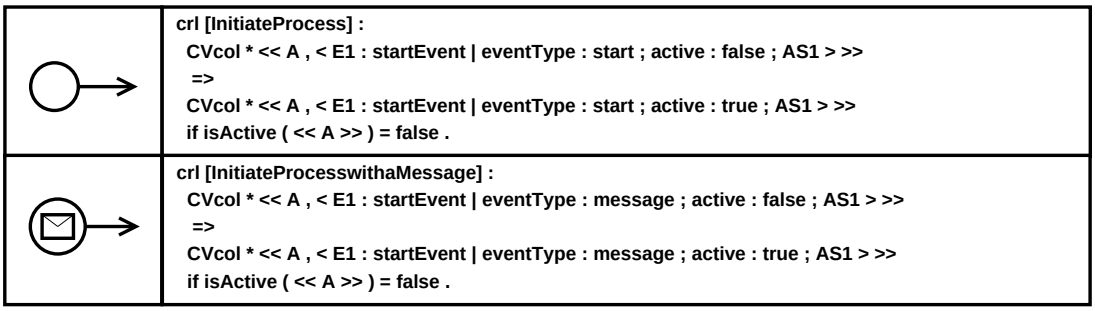


Figure 3.11: Process Initiation Rules

In Figure 3.11, the rules are presented; i.e. if the start event is inactive, the rule rewrites it to be active after checking that nothing else is active in the process (`isActive` retrieves *false*). The second rule in the figure assumes the start of a process is initiated by receiving a message (i.e. message start event). The initiation procedure is the same in both cases. A process can start as a result of receiving a message (e.g. receiving an application form or a request). In this case the message event initiates the process. Hence we have the rule `InitiateProcesswithaMessage`. This rule, in Figure 3.11, acts like the normal initiation rule for the process `InitiateProcess` except that the type of event is different.

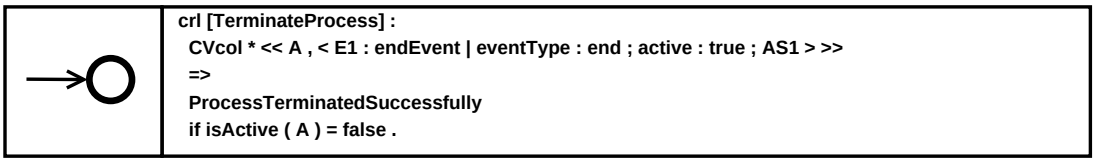


Figure 3.12: Process Termination Rule

In the other side of the process, the rule `TerminateProcess` terminates a process if it is active and the only active object is the end object. It changes the process state from active to inactive. The rule is applicable when the process is in its end state S_e (c.f. Definition 3.4.3). It rewrites the object set into a descriptive statement indicating process termination (e.g. the term zero). The function `isActive` should retrieve *false* here as a condition of application.

Sequential Behaviour

An object is active if its attribute `active` is *true*. For example, in Figure 3.13, in a certain state of the process, the activity `X` is active while the activity `Y` is not active. Simulating process behaviour can be thought of as if the activation is passed from one object to its immediate successor(s) if certain conditions satisfied in come cases. The next state according to the rule in Figure 3.13 is where `X` inactive and `Y` is active. The condition that should be fulfilled here is the activity `Y` should be the immediate successor for the activity `X`. In a more general case, the successor can be another object (e.g. gateway, activity, event, ... etc). Hence we can use the rule `Seq` (in Figure 3.13) to simulate the sequential behaviour of objects can have in a process. The `X`, and `Y`, are of sort `Oid`, `K` and `L` are variables typed with the sort `Cid`, `t N1` is the transition linking the two objects, `AS1` and `AS2` representing the rest of the attributes that an object might have, and `A` is the remaining objects in the process.

Sequential processes may have conditions restricting their execution, such as waiting for a message to arrive or a data object to be updated. This situation can be modelled as an execution precondition, in which the object is specified as *Ready to be active* but not actually active using the attribute `ToBeActive`. At the same time, the attribute `cond` indicates this condition case, where it has the value *true* if there is an execution condition for this object, or *false* if there is no attached conditions. Therefore, the rule `Seq-Cond`

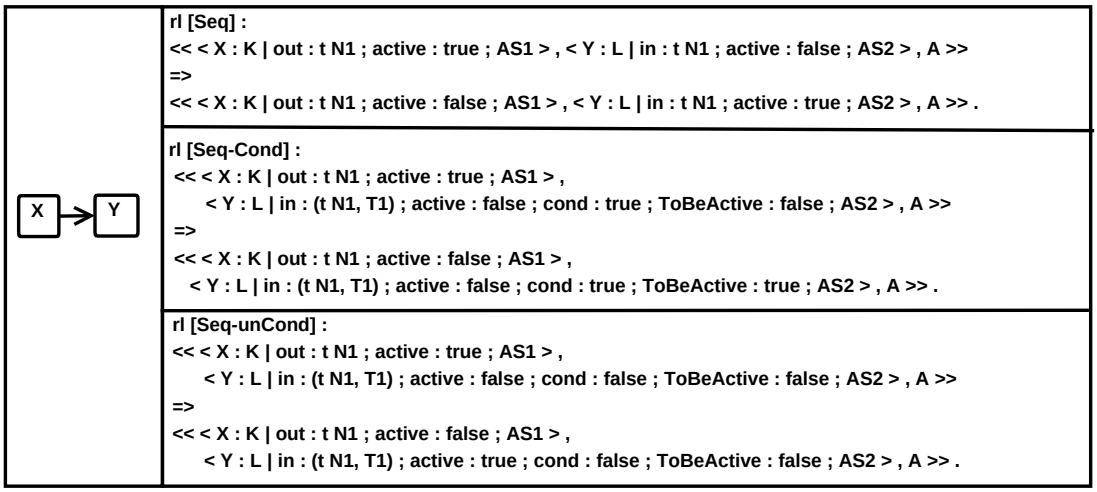


Figure 3.13: Sequence Rule

in Figure 3.13 requires the object `Y` to have the value *true* in its attribute `cond`, and hence will transform its state to *ready to be active* state with a value *true* for the attribute `ToBeActive`. In case of unconditional sequential execution of objects, the attributes `cond` and `ToBeActive` should be included with value *false* in both sides of the rule as shown in rule `Seq-unCond` in Figure 3.13. We use the rules `Seq-Cond` and `Seq-unCond` in our semantics and we removed rule `Seq` from the Maude code as it represents a too general case which we can handle using one of the other two rules.

Parallel Behaviour

Fork gateway is used in the BPMN to refer to the dividing of a path into two or more parallel paths. When a fork gate is active, the rule `ANDfork` can be applied to activate all immediate successor objects.

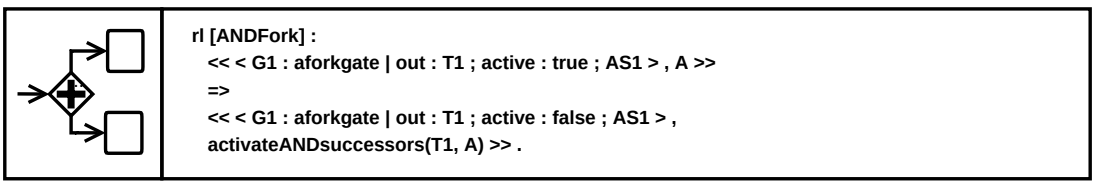


Figure 3.14: Parallel Fork Rule

This rule fetches all the successors of the gateway and activates them concurrently and deactivates the `aforkgate` gateway afterwards. In the rule above (in Figure 3.14), `G1` is a variable for the object identifier, `T1` is a set of transitions that connects the objects to

the gateway. Activating the immediate successors of the fork gateway is achieved using the function `activateANDsuccessors` which takes the output transitions for the fork gateway and the rest of the object set and produces the same object set after activating the immediate successors. The order of the activation does not matter as they are supposed to be executed concurrently with no such restriction. The definition of the function is in Appendix B.

In case of joining parallel activities, the `ajoingate` cannot be activated until all it predecessors have finished (deactivated). It can happen that the predecessor objects executed and finished in different times, i.e. in Figure 3.15, activity `b` may finish before activity `c` finish, then following the `Seq-Cond` rule, the AND join gateway should be active. However, according to the AND join semantics, it should wait for all its immediate predecessors to be executed. Therefore, we mark the attribute `cond` to be `true` and the attribute `ToBeActive` to `false` in its initial state. After that, the first immediate predecessor to finish causes the attribute `ToBeActive` to be changed to `true`, however, the gateway itself is still not active.

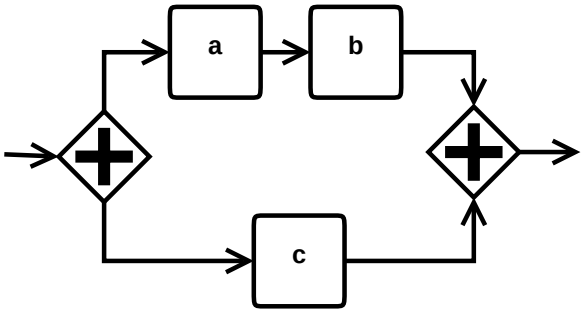
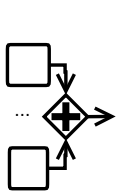


Figure 3.15: Example model with AND fork and join gateways

As explained above in Section 3.4.2, if there are execution conditions for the object, then its `cond` attribute will be *true*. In the case of AND join gateway, the trigger for making it in the *ready to be active* state is having at least one active predecessor which then will rewrite its attribute `ToBeActive` from *false* to *true* as described in the rule `Seq-Cond-ajoin` in Figure 3.16.

The `ANDJoin` rule activates the join gateway and then the immediate predecessors are fetched and deactivated using the functions `activePreds`. The function checks if there are any active objects in the gateway upstream (i.e. objects between the AND fork and the AND join gateways) using the function `preds` to fetch the predecessor



```
crl [ANDJoin] :
 << < G1 : ajoingate | in : T1 ; itsSplit : G2 ; active : false ; cond : true ;  ToBeActive : true ;
 AS1 > , < G2 : aforkgate | AS3 > , A >>
=>
 << < G1 : ajoingate | in : T1 ; itsSplit : G2 ; active : true ; cond : true ; ToBeActive : false ;
 AS1 > , < G2 : aforkgate | AS3 > , A >>
if activePreds(T1, preds(< G2 : aforkgate | AS3 > , < G1 : ajoingate | in : T1 ; itsSplit : G2 ;
 active : true ; cond : true ; ToBeActive : false ;  AS1 >,
 (A, < G2 : aforkgate | AS3 >), noobject, noobject)) == false /\
 isActive ( << imPreds (T1, A, noobject) >> ) == false .
```

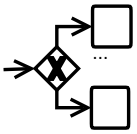
```
rl [Seq-Cond-ajoin] :
 << < X : K | out : t N1 ; active : true ; AS1 > , < Y : ajoingate | in : (t N1, T1) ; active : false ;
      cond : true ; ToBeActive : true ; AS2 > , A >>
 =>
 << < X : K | out : t N1 ; active : false ; AS1 > , < Y : ajoingate | in : (t N1, T1) ; active : false ;
      cond : true ; ToBeActive : true ; AS2 > , A >> .
```

Figure 3.16: Parallel Join Rule

objects and then checks if they are active[9]. The AND fork and join gateways are linked through the attribute `itsSplit` in the join gateway, which refer to the corresponding fork gateway. The second condition is to ensure that by this stage in the execution all the immediate predecessors for the join gateway are inactive using the functions `imPreds` and `isActive` respectively.

Exclusive Decision-Based Behaviour

Exclusive data-based decision gateways (XOR-split) is designed to choose only one alternative to activate among its immediate successors [68]. The rule `XORsplit` simulates the behaviour of the split gateway and described in Figure 3.17.



```
rl [XORsplit] :
  (CVs1) * << < G1 : xsplitgate | out : T1 ; defaultFlow : T2 ; guard : GExp1 ; error : t N2 ;
  controlValues : noControlValue ; active : true ; AS1 > , A >>
 =>
  (CVs1) * << < G1 : xsplitgate | out : T1 ; defaultFlow : T2 ; guard : GExp1 ; error : t N2 ;
  controlValues : assignCVs(GExp1, CVs1) ; active : false ; AS1 > ,
  activateXORchild (T1, T2, t N2, GExp1, assignCVs(GExp1, CVs1), A) >> .
```

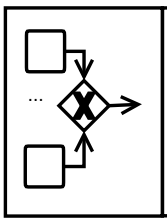
Figure 3.17: Exclusive Data-based Decision (XOR) Split Rule

In order to determine the outgoing object that is going to be activated, all the conditions are evaluated as long as their control values have been passed through to the gateway during the execution. Control values (`CVs`) are used to provide information to the gateways to be able to evaluate the guard expressions and process branching

[9]The definition of the function is in Appendix B

decisions. The function `activateXORchild` is used to activate the successor object that its guard expression evaluated to *true* in the evaluation function. The function `activateXORchild` definition is in Appendix B and the function `assignCVs` is defined later in this section.

For the merging behaviour of the exclusive gateways, the rule `XORmerge`, activates an `xmergegate` object if it has one active immediate predecessor, and it deactivates this predecessor, as shown in Figure 3.18. The application of this rule requires an active predecessor to an inactive XOR-merge gateway. Then the activation is toggled and the process is ready for another rule application, which is normally be the sequential rule to activate the XOR-merge successor.



```
crl [XORmerge] :
 << < X : K | out : (t N1, T2) ; active : true ; AS1 > , < G1 : xmergegate | in : (t N1 , T1) ;
  active : false ; AS2 > , A >>
 =>
 << < X : K | out : (t N1, T2) ; active : false ; AS1 > , < G1 : xmergegate | in : (t N1 , T1) ;
  active : true ; AS2 > , A >>
 if K =/= xsplitgate .
```

Figure 3.18: Exclusive Data-based Decision (XOR) Merge Rule

Notice the condition in the rule excludes the case of having a XOR split predecessor to the XOR merge gateway. That is to allow the guard evaluation in the XOR split gateway to take place and hence activating the right XOR child. For example, in case of direct flow from a split gateway to a merge gateway as in Figure 3.19 where the flow `t3` coming from the XOR split gateway and entering the XOR merge gateway which may cause the activation of `g2` before `g1` decides which branch to choose. At the same time, it can happen that task `a` is executed and at the same time, `g2` is activated, which will end up in the XOR merge gateway `g2` being executed twice (a lack of synchronization situation) in the process.

Inclusive Decision-Based Behaviour

In BPMN, inclusive decision (OR) gateway is used to choose one or more branches at the same time. In our formalization, the constructor operators *osplitgate* and *omergegate* are used for OR split and merge gateways `Cid` respectively. For inclusive decision-based pattern of behaviour, the OR gateway represents the ability to activate one *or*

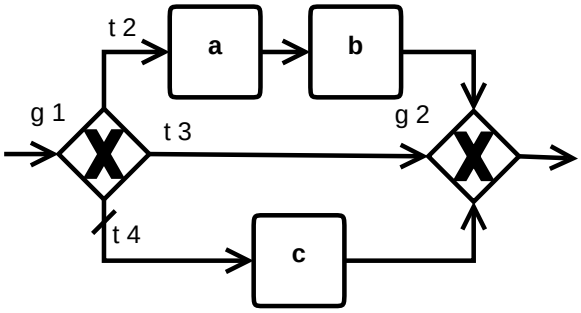


Figure 3.19: Example model with XOR split and merge gateways

more branches at the same time. Hence, the true evaluation of one condition expression does not exclude the evaluation of other condition expressions. In the splitting behaviour, the rule `ORsplit` is used to activate the OR-split successor(s) (using function `activateORchildren`) according to the evaluation of the guard expressions in the gateway.

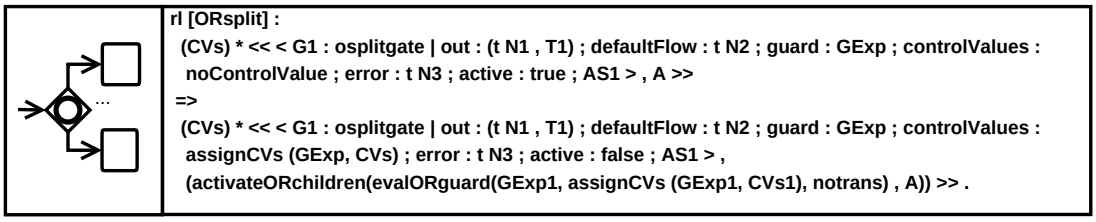


Figure 3.20: Inclusive Decision (OR) Split Rule

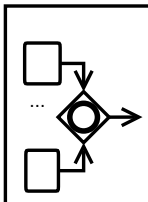
The rule `ORsplit` is applicable when the OR split gateway is *active*, then the gateway's `controlValues` attribute value will be rewritten to the control values that are contained into the input set (`CVs`), the rest of the objects in the `ObjectSet` (i.e. `A`) are rewritten using the function `activateORchildren`, and finally the split gateway itself is deactivated (`active:false`). The function `evalORGuard` helps the gateway to decide which successors to activate as discussed later in this Section. The function `activateORchildren` is used to activate all OR successor objects which have incoming transition listed in the set of active transitions in its first argument (`TransSymbol`). When it activates them all, it retrieves the full set of objects `A`, which now has all the active OR-split immediate successors.

```
op activateORchildren : TransSymbol ObjectSet -> ObjectSet .
eq activateORchildren (notrans, A) =  A .
eq activateORchildren ((t N1,T1),
```

```
(< X : K | in : t N1 ; active : false ; AS1 >, A))
= activateORchildren (T1,
(< X : K | in : t N1 ; active : true ; AS1 >, A)) [owise] .
```

The merge OR gateway behaviour is not like an AND-join, which knows for sure that all its predecessors are active objects and should be deactivated in order to give the AND-join gateway the activation mode. Moreover, its behaviour is not like an XOR-merge which knows for sure that only one active predecessor is there and it has to be deactivated before activating the gateway. However, in the case of a OR-merge gateways, how could a certain gateway decide if the coming flows are all chosen (activated) by its OR-split gateway? Notice, the number of active predecessor flows is unknown to the OR-merge gateway. Therefore, there should be a link that relates a merge gateway to its split gateway in the same block; in order to keep track of the number of activated flows that need to be deactivated as an activation condition for the merge gateway. To relate the two gateway objects, an attribute `itsSplit` is defined in the merge gateway which pass this piece of information to the merge gateway. It is defined in the merge gateway because it is the point in the process flow that should consider collecting and combining all the split flows from a certain OR-split gateway. In the specification below, the rule `ORmerge` rewrites the OR merge gateway object state from being *inactive* to being *active* after satisfying the condition that the gateway is ready to merge (function `Ready2Merge` evaluates *true*). At the same time, the function `deactivateORpreds` deactivates the immediate predecessors of the merge gateway as a result of activating the OR merge gateway. This is to make sure that all the activated flows have been synchronized at this point in the flow and no OR immediate predecessor object is left active. If this function is not used, a lack of synchronization can result, i.e. there will be a situation when the OR merge gateway is activated more than once as the activation is passed to it from more than one predecessor, possibly in different points of the time.

The function `Ready2Merge` is used to check if all the activated immediate predecessors are active and ready for the merge. This is done by retrieving *true* when it counts the number of activated branches by the corresponding OR split gateway and finds the same number of activated immediate predecessors to the OR merge gateway. Here, the relation between the two gateways as one block facilitates the efficient execution and evaluation of the guards. It enables the merge gateway to know how many branches



```
crl [ORmerge] :
 << < G1 : omergegate | in : (t N1 , T1) ; itsSplit : G2 ; active : false ; AS1 > ,
 < G2 : osplitgate | active : false ; guard : GExp ; controlValues : CVs ; AS2 > , A >>
=>
 << < G1 : omergegate | in : (t N1 , T1) ; itsSplit : G2 ; active : true ; AS1 > ,
 < G2 : osplitgate | active : false ; guard : GExp ; controlValues : CVs ; AS2 > ,
  deactivateOPreds((t N1, T1), A) >>
 if Ready2Merge(evalORguard (GExp, CVs, notrans), (t N1, T1), A ) == true .
```

Figure 3.21: Inclusive Decision (OR) Merge Rule

have been activated after its corresponding OR-split gateway.

```
op Ready2Merge : TransSymbol TransSymbol ObjectSet -> Bool .
eq Ready2Merge((notrans), (T1), A) = true .
eq Ready2Merge((t N1,T1), (t N2,T2),
  (< X : K | out : t N2 ; active : true ; AS1 >, A))
 = Ready2Merge((T1), (T2),
  (< X : K | out : t N2 ; active : true ; AS1 >, A)) [owise] .
```

It takes the set of OR split gateway successor transitions, the set of the OR merge input transitions and the whole set of objects. In the rule `ORmerge`, in Figure 3.21, this boolean function is used to take the activated transitions resulted from the evaluation function `evalORguard` and checks if the same number of predecessors is active and ready to be merged. This prevents the occurrence of a deadlock situation when the merge gateway is waiting for an object to be activated while it is not in the execution path at all. Moreover, it facilitate the processing of nested OR gateways by marking each merge gateway with its OR-split companion avoiding mixing the evaluated conditions from different gateways.

In Rule `ORmerge`, the OR-merge gateway is activated and its immediate predecessors are deactivated after fulfilling the condition of being `Ready2Merge`. The function `deactivateOPreds` is used to deactivate the OR-merge immediate predecessors following the definition in Appendix B. A detailed analysis of the deadlock situation resulting from the improper use of OR gateways (e.g. missed flows at the merge gateway, a gateway is waiting for an inactive flow which will never be active, and gateways with lack of synchronization) is discussed in Chapter 4. In the following section, the guard expression evaluation mechanism is introduced and discussed. It is worth describing the guard definition and evaluation mechanism as *novel* in the context of BPMN business process

formalizations.

Evaluating Guard Expressions

Decision based gateways control the flow in BPMN processes. In our formalization, decision-based (split) gateways hold the guard expressions which direct the flow in the process based on the expressions evaluation. Using the CFG defined in Section 3.1.3, the guard expressions are set for evaluation. The evaluation is done through the function `evalGuard` which take as inputs the gateway guard expression (which is normally one expression), the set of control values (discussed in Section 3.4.2), and produces a set of transitions referring to the branch associated with the successfully evaluated expression. For example, if the expression uses the equality operator (`==`), then the condition should be checking if the two operands are equal using the equality definition in module `NAT` if the operands are numbers and in module `STRING` if the operands are String characters. For such reasons we use the command (`protecting NAT STRING`) at the beginning of the Maude module.

```
op evalGuard : Gexp ControlValue TransSymbol -> TransSymbol .
eq evalGuard(noexp, noControlValue, T1) = T1 .
ceq evalGuard(((V1 == N1,tN2).GExp),((V1:N3)..CVs),T1) = tN2
 if N3 == N1 .
eq evalGuard(((V1 == N1,tN2).GExp),((V1:N3)..CVs),T1)
 = evalGuard(((V1 == N1,tN2).GExp),CVs,T1) [owise] .
ceq evalGuard(((V1 == S1,tN1).GExp),((V1:S2)..CVs),T1) = tN1
    if S2 == S1 .
eq evalGuard(((V1 == S1,tN1).GExp),((V1:S2)..CVs),T1)
 = evalGuard(((V1 == S1,tN1).GExp),CVs,T1) [owise] .
```

The function `evalGuard`, defined above, takes the information needed to decide on which successor object should be activated next from the control values. For example, in Figure 3.7, the XOR-split gateway g_1 decides if the authorization is granted or declined to modify the baselines, therefore, the gateway has two outgoing transitions (t_3,t_4), the guard expressions are `((Authorized?=="YES",t3).(Authorized?=="NO",t4))`, and the control value for this case is `(Authorized?:"YES")` then the function retrieves `t3` as the string control value (i.e. `YES` in attribute `controlValue`) *equals* the String

value in the guard expression (i.e. YES in attribute guard). Here, the modeller is responsible for assuring that the expressions are mutually exclusive, however, the semantics of an exclusive split gateway requires evaluation of the guard expressions one by one and if an expression is successful, then no more expressions are evaluated, as it is only one alternative to be chosen. If the expressions are not mutually exclusive for the XOR split gateway guard, there is a possibility that the first expression evaluating to *true* is the one considered in deciding the activation of the successor object, leaving another possible alternative remains unknown to the process. The same function is defined to evaluate guard expressions with other logical comparison operators (==, =/=) for Strings and (=/=, <=, >=, <, >) for numbers (c.f. the CFG defined in Section 3.1.3) as detailed in Appendix B.

In the case of OR split guard expression evaluation, the function evalORguard is introduced. The OR split semantics requires all the conditions to be evaluated and, as a result, can have as many output transitions as the OR-split outgoing transitions. Therefore, the function evalORguard takes the following arguments: the guard expressions (GExp) and the assigned control values resulting from (assignCVs(GExp,CVs)) and retrieves the set of transitions whose associated guard expression evaluates to *true*.

```
op evalORguard : Gexp ControlValue TransSymbol -> TransSymbol .
eq evalORguard(noexp, noControlValue, T1) = T1 .
eq evalORguard(((V1 == N1,tN2) . GExp), ((V1:N1)..CVs), T1)
   = evalORguard(GExp, CVs, (tN2,T1)) .
ceq evalORguard(((V1 == N1,tN2) . GExp), ((V1:N3)..CVs), T1)
   = evalORguard(((V1 == N1,tN2) . GExp), CVs, T1)
   if N1 =/= N3 .
eq evalORguard(((V1 == N1,tN2) . GExp), CVs, T1)
   = evalORguard(GExp, CVs, T1) [owise] .
```

The definition above of the function evalORguard defines the equality relation over the numeric values in guard expressions. A similar set of equations can be used for the equality relation over the string values in guard expressions by changing the sort type to STRING for String characters. The evaluation considers the control values to match with the variable values in the guard expressions. In the next section, the control values are explained.

Managing Control Values

In the formalization we presented so far, deciding which flow coming out from a split gateway depends on the control values assigned to the attribute `controlValues`. However, they are not yet automated in the formalization. The control values are normally entered by the process modeller in the design time for each gateway. Trying to minimize the time and effort of managing the control values for large BPMN models, the following mechanism is introduced using the configuration (_*_), where the first underscore is substituted by a collection of control values and the second underscore with the well-formed process `W-BPMN`. The general form is defined as follows:

```
subsort ControlValue < CVcollection .
subsort ObjectSet < TraceObjectSet .
op _*_ : CVcollection ObjectSet -> TraceObjectSet .
```

The resulted configuration is considered one possible trace for the model. Thus we can consider all possible traces for the model if we listed the set of all possible collections of control values for a certain model. The process traces are possible execution paths for the business process (c.f. Definition 3.4.5). In the process, the split gateway object has the attribute (`controlValues`) assigned the value `noControlValue` during the design time. This value should be changed to the chosen control values during the process execution. There can be more than one split gateway in the process, and therefore more than one control value for the single execution (instance) of the process. The collection of control values needed for a single process instance is defined as an associative set of control values separated by the operator (`,`).

```
op _,_ : CVcollection CVcollection -> CVcollection
                                    [ctor assoc id: noControlValue] .
```

On automating the procedure of assigning the control values to automatically generate the traces, our proposed Maude based tool does this automatically. The idea is to extract the guard expressions from the split gateways (i.e. XOR and OR) and then create the corresponding control values for each guard expression. This is conducted using the operators `guardExtract` and `createCV` respectively. For the first operator, we provide part of the definition below. The arguments are the object set representing the process and an initially empty set of guard expressions. Once a split gateway is found (e.g. XOR split here), the guard expressions are copied to the set of output guard expressions. This

happens until no more split gateways in the process, then the operator returns the set of collected guard expressions.

```
op guardExtract : ObjectSet Gexp -> Gexp .
eq guardExtract(A , GExp1) = GExp1 .
eq guardExtract(CVs1 * << < G1 : xsplitgate | guard : GExp1 ;
                AS1 > , A >> , GExp2)
 = guardExtract(CVs1 * << < G1 : xsplitgate | guard : GExp1 ;
                AS1 > , A >> , (GExp1 . GExp2)) [owise] .
```

After that the control values are created from the extracted guard expressions using the operator `createCV` defined below for the equality of string and numeric values as example of the implementation.

```
op createCV : Gexp ControlValue -> ControlValue .
eq createCV (noexp , CVs1) = CVs1 .
eq createCV (((V1 == S1 , t N1) . GExp1) , CVs1)
 = createCV (GExp1 , (CVs1 .. (V1 : S1))) [owise] .
eq createCV (((V1 == N1 , t N1) . GExp1) , CVs1)
 = createCV (GExp1 , (CVs1 .. (V1 : N1))) [owise] .
```

There is still one thing missing, which is how these different control values will be assigned to the corresponding gateways during simulating process execution. For that, the function `assignCVs` is defined to take the collection of control values and assign them to their corresponding split gateway object attribute `controlValues` in the process. This matching is performed using a combination of the control values and the guard conditions in the gateways. The function specifications for the equality guard condition in case of numeric and string values are:

```
op assignCVs : Gexp ControlValue -> ControlValue .
eq assignCVs(noexp,CVs1) = noControlValue .
eq assignCVs(((V1==N1,tN2).GExp1),((V1:N3)..CVs1)) = V1:N3 .
eq assignCVs(((V1==S1,tN1).GExp1),((V1:S2)..CVs1)) = V1:S2 .
```

The BPMN model which contains split gateways should now look like the example in Figure 3.7. In particular, we want to show the process of assigning the control values and how this is contributing in evaluating the guard expressions. The Maude code in

Figure 3.8 represents the BPMN diagram of the model with the collection of control values. The resulting configuration is one possible trace for the model. In this case, by simulating the process execution, it will result in the set of all possible traces for the model with respect to gateway routing, i.e. `((CVcol1 * A),(CVcol2 * A),...)` where `CVcol1,CVcol2,...` are the possible collections of control values for the process `A` created using the above mechanism. As an example of an output execution path for the process model in Figure 3.7 can be represented as:

```
((Authorized?:"YES"),(AuthorizedCR?:"YES"),(MoreCR1?:"NO"),
(MoreCR2?:"NO")) * ReleaseBaseline
```

where `Authorized?`, `AuthorizedCR?`, `MoreCR1?`, and `MoreCR2?` are defined as variable names and `YES`, `YES`, `NO` and `NO` are possible values specifying access granted, authorized change request, no more change requests to choose from, and no more change requests to apply respectively.

3.4.3 Exception Handling

An exception is activated if the exception error value in the activity it is attached to is evaluated to *true*. The exception error attribute `excValue` has a default value of *false*, and it can be changed to *true* as a user input value for simulating the process behaviour in this particular case (i.e. firing the exception). In this case, the rule `ExceptionHandling` will be applied to pass activation to the exception rather than the normal flow objects.

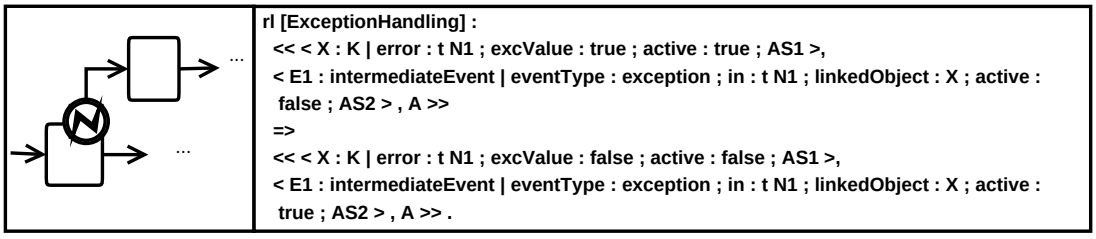
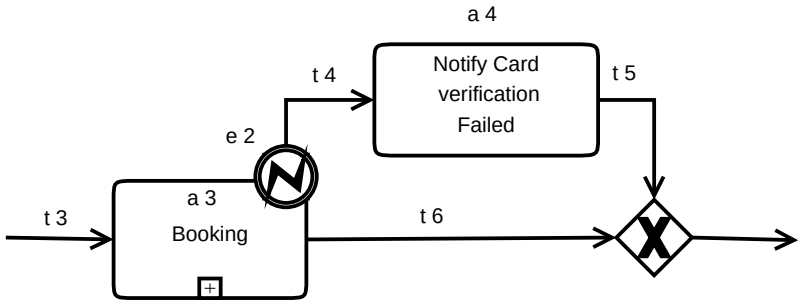


Figure 3.22: General Exception Rule

In Figure 3.22, the rule `ExceptionHandling` is simulating the general exception behaviour where the exception is attached to the boundary of the object `X`. The attribute `excValue` value indicates that the exception situation is active (i.e. *true*), the activation is passed to the exception attached to it, with identifier `E1`. In Maude representation, objects like `X` (i.e. with boundary attached events) are linked to these event objects explicitly

via the attribute `linkedObject` in the corresponding event. Exceptions can also occur as part of the gateway behaviour if no flows are available to pass the activation to due to lack of information (control values/conditions). In this case, a gateway exception object is activated. This can happen with an OR or XOR split gateways. Such situation has been considered when designing the `activateXORchild` and `activateORchildren` functions.



< a 3 : subprocess | name : "Booking" ; in : t 3 ; out : t 6 ; error : t 4 , ... > ,
< e 2 : intermediateEvent | eventType : exception ; in : notrans ; out : t 4 ; linkedObject : a 3 > ,
< a 4 : task | name : "Notify Card Verification Failed" ; in : t 4 ; out : t 5 , ... >

Figure 3.23: Example for Exception handling with Maude representation syntactically

To illustrate how our formalization deals with exceptions attached to activity boundary, the example is shown in Figure 3.23. It is part of a model for an airline system (discussed in [104]), where the client choose the seats before proceeding in the booking process. In case the client payment card did not validated, the system raises an exception and notify the client of the reasons. The process then continues, however, we only use this part to clarify the exception representation in the formalization. In our formalization, the number of incoming flows for a merge gateway should be equal to or more than the number of outgoing flows for its corresponding split gateway. From a semantic point of view, an exception can take place in one of the split flow paths producing an extra flow. This require the `exception flow` to be connected to the corresponding merge gateway afterwards (e.g. with XOR merge gateway as in Figure 3.23).

3.4.4 Message Handling

Messages are intermediate events used to connect the flow elements in different pools, where sequence flow cannot be used. For example, the communications between the

purchaser and their supplier can be represented by message events. Each message connects two activities in two different pools. In order to simulate the role of messages in BPs, we model the messages as objects in the process object set. Hence it can be activated as a sign of having information to deliver. It can also restrict the activation of the linked object, which will be at that time waiting for some information to be delivered in that message to continue the process execution. A message object does not have a status to be changed during the execution, however, it still has the active attribute which define its state as active or not active object.

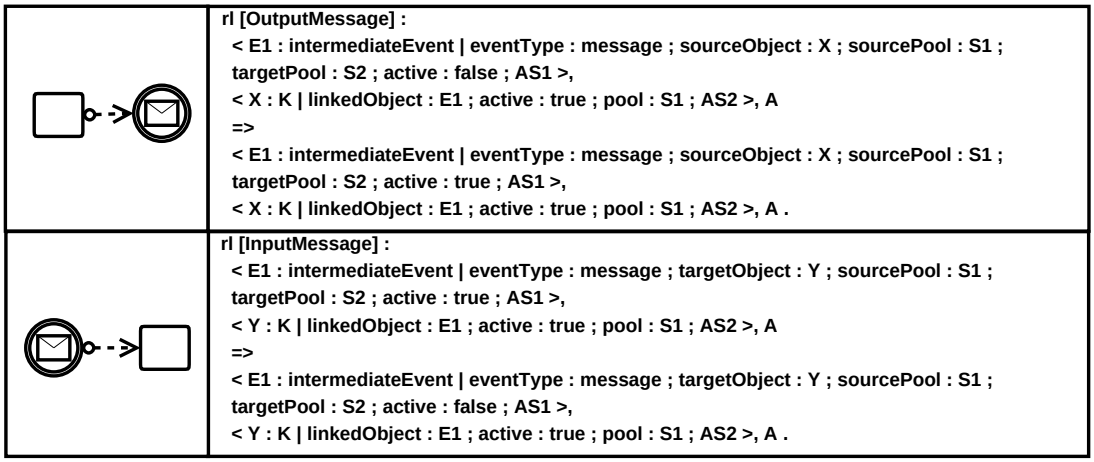


Figure 3.24: Message Handling Rules: input and output messages

In the rule `OutpurMessage` in Figure 3.24, the message source activity is active, then the rule activates the message itself. However, the rule keeps the activity active to allow for the normal flow to take place. In opposite, in the rule `InputMessage` in Figure 3.24, the message is active and its target object is active too, then the rule deactivate the message while leaving the other object for the normal flow rules to take place. Notice that attributes `sourcePool` and `targetPool` specifies the pools from which the message is sent and is received. These two pools should be different according to the well-structured business processes discussed in Definition 3.3.2.

3.4.5 Subprocess Semantics

A sub-process may contain other objects (i.e. events, activities, gateways and objects). In this formalization we consider a dummy start and end events for starting and ending a sub-process with the types `startSubprocess` and `endSubprocess`. A sub-process is

ready for initiation if its intermediate predecessor is active, and it is initiated by activating its dummy start event, as shown in rule `enterSubprocess` in Figure 3.25.

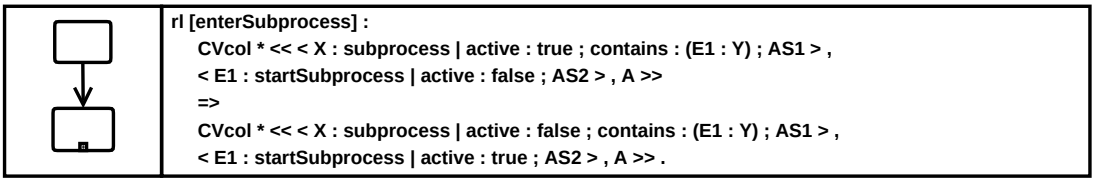


Figure 3.25: Enter Sub-process Semantic Rule

By completing the sub-process and while the dummy end event is active, the rule `TerminateSubprocess` deactivates the dummy end event (i.e. sub-process is terminated) and activates its successor as shown in Figure 3.26.

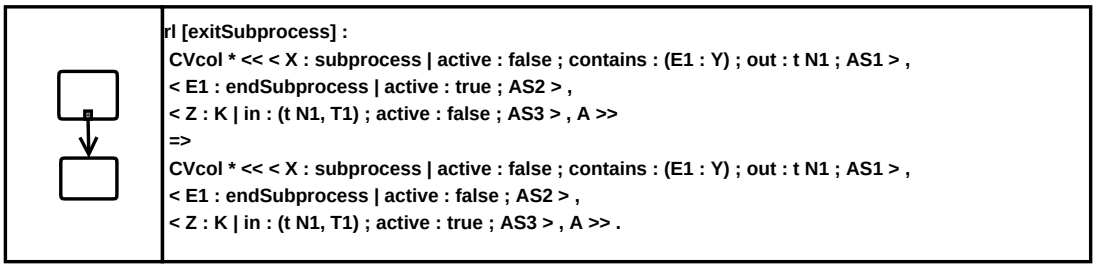


Figure 3.26: Terminate Sub-process Rules

3.4.6 Data Handling

Data objects represents all sorts of documents that are moving around in the organization and used by a BP. As described in Section 3.1, a data object has a status which changes as an effect of the activities using it. The semantics of the data objects behaviour in our approach is specified using the rules in Figure 3.27. In `DataOutput`, the data object is an output for the activity `X`. The link between the activity and the data object is formed from the attribute `linkedObject` in the data object and the attribute `hasOutput` in the activity. In `DataInput`, the data object is an input for the activity `X`. The link between the activity and the data object from the attribute `linkedObject` in the data object and the attribute `hasInput` in the activity.

The existence of data objects and other constructs in the BP adds restrictions to its execution. These restrictions are important to be modelled in the design time where

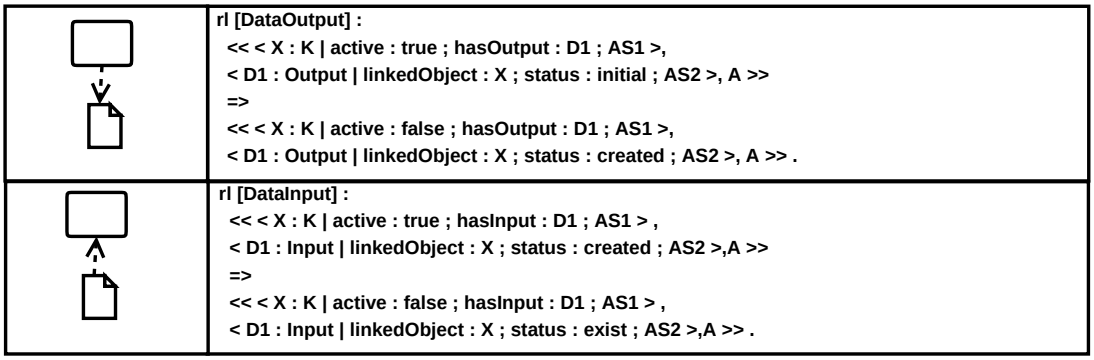


Figure 3.27: Data Object Handling Rules: input and output data objects

the process is more flexible to be modified than to enforce them in the implementation and deployment phase. In the following section, we introduce some domain specific semantic rules for the example introduced in Section 3.2.

3.4.7 Domain Specific Semantics

Each BP represents a specific work procedure carrying its characteristics, conditions and constraints. This can be shown by specific patterns of behaviour, assigned to process elements, which are dependent on the corresponding business environment or other elements in the same process. In the proposed example in Figure 3.7, tasks like `Retrieve CI` or `Change CI` should be active only if a change request (`CR`) is open. Another example exists when a data object `Baseline` status should be changed to *released* if the task `Release Baseline` is active (i.e. a new baseline is released). Such behaviour requires specific rules which are considered related to the domain of the business process under consideration. We call these rules *Domain Specific Rules (DSR)*, which can be different from one process to another. For the illustrated example in Section 3.2, we provide the following DSRs as an example of the formalization validity in domain specific requirements for business processes. In Figure 3.28, a set of rules is modelled graphically for presentation purposes while the corresponding term rewrite rules are coded in Maude in Figure 3.29 and Figure 3.30 using the syntactical notation presented earlier in this chapter. In Figure 3.28, the black dot on a task indicates that this task is active and the white dot on a task indicates that this task is inactive.

To start with, the possible status (defining their data object lifecycle [102]) for a Change Request (CR) data object are: *initial*, *open* and *closed*; for a Configuration Item

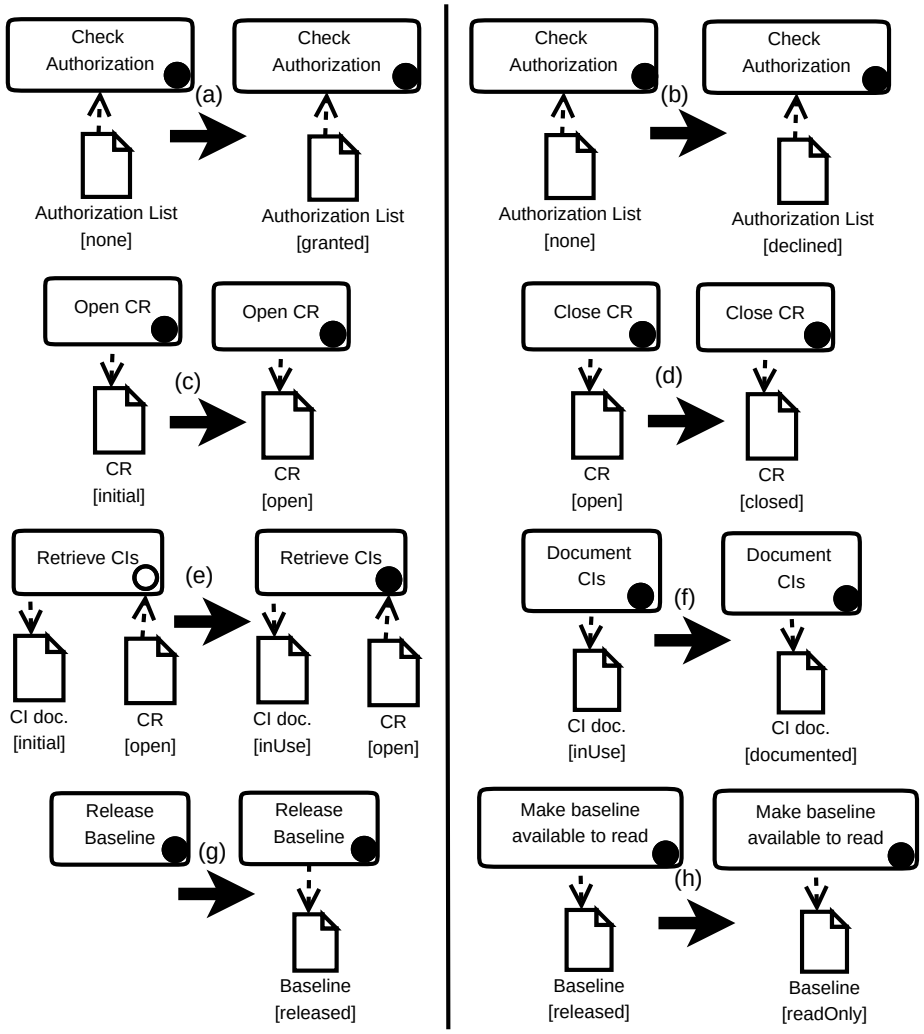


Figure 3.28: Domain-Specific Rewrite Rules

(CI doc) data object are: *initial*, *inUse* and *documented*; for Authorization List data object are: *granted* and *declined* and for a Baseline data object are: *none*, *released* and *readOnly*. In Figure 3.28, some rules are applied when the corresponding task is active (marked by the black circle) such as `Change CI` and others are dependent on the status of a connected data object, like `Retrieve CI`.

In the example (c.f. Section 3.2), an authorization is required in order to release the baselines. The rule (a) (*Grant Authorized*) checks if the access is being done by an authorized role listed in the `Authorization List` and then changes the data object status to *granted*. The rule uses the control value (`Authorized?:"YES"`) to grant permission to change and release the baseline. If the access information is not listed in the , or if it is not listed in the document `Authorization List`, then the data object status will

```
(a) Access Authorized
 rl [GrantAuthorization] :
  ((Authorized? : "YES") .. CVs) * << < X : K | name : "Check Authorization" ; active : true ; AS1 > ,
  < D1 : dataobject | name : "Authorization List" ; status : none ; AS2 > , A >>
=>
 ((Authorized? : "YES") .. CVs) * << < X : K | name : "Check Authorization" ; active : true ; AS1 > ,
 < D1 : dataobject | name : "Authorization List" ; status : granted ; AS2 > , A >> .

(b) Access Declined
 rl [DeclineAuthroization] :
 ((Authorized? : "NO") .. CVs) * << < X : K | name : "Check Authorization" ; active : true ; AS1 > ,
 < D1 : dataobject | name : "Authorization List" ; status : none ; AS2 > , A >>
 =>
 ((Authorized? : "NO") .. CVs) * << < X : K | name : "Check Authorization" ; active : true ; AS1 > ,
 < D1 : dataobject | name : "Authorization List" ; status : declined ; AS2 > ,  A >> .

(c) Open a Change Request
 rl [OpenCR1] :
 CVs * << < X : K | name : "Open CR" ; active : true ; hasOutput : D1 ; AS1 > ,
  < D1 : dataobject | name : "CR" ; status : initial ; AS2 > , A >>
 =>
 CVs * << < X : K | name : "Open CR" ; active : true ; hasOutput : D1 ; AS1 > ,
  < D1 : dataobject | name : "CR" ; status : open ; AS2 > , A >> .

(d) Close a Change Request
 rl [CloseCR1p5] :
  CVs * << < X : K | name : "Close CR" ; active : true ; hasOutput : D1 ; AS1 > ,
  < D1 : dataobject | status : open ; AS2 > , A >>
 =>
  CVs * << < X : K | name : "Close CR" ; active : true ; hasOutput : D1 ; AS1 > ,
  < D1 : dataobject | status : closed ; AS2 > , A >> .
```

Figure 3.29: Maude Representation for DSR from (a) to (d) in Figure 3.28

change to declined as specified by rule (b) (*Decline Authorization*).

For each change request (CR) enters the process, its status should change from `initial` to `open` to indicate its use. In Figure 3.29, rule (c) (*Open a Change request*) is simulating this behaviour. After the change request is used and changes applied, it needs to be closed. The status of the data object is changed from `open` to `closed` as declared in rule (d) (*Close a Change Request*).

While the change request is open, the corresponding configuration item CI is retrieved. In Figure 3.7, the activity `Retrieve CI` is activated if a change request is open, i.e. the CI is retrieved to be processed. This is modelled using the rule (e) in Figure 3.28 and rule *Retrieve CI* in Figure 3.30. At the same time, the retrieved CI status should be changed to indicate that it is in use (i.e. change its status from `initial` to `inUse`) as described in the same rule (rule (e) in Figure 3.28) and rule *Using CI* in Figure 3.30.

```
(e) Retrieve and Use CI
 rl [RetrieveCIp2] :
  CVs * << < X : task | name : "Retrieve CI" ; in : t N1 ; cond : true ; ToBeActive : true ;
  active : false ; hasInput : D1 ; AS2 > , < D1 : dataobject | status : open ; AS3 > , A >>
  =>
  CVs * << < X : task | name : "Retrieve CI" ; in : t N1 ; cond : true ; ToBeActive : false ;
  active : true ; hasInput : D1 ; AS2 > , < D1 : dataobject | status : open ; AS3 > , A >> .

 rl [UsingCIp3] :
  CVs * << < X : task | name : "Change CI" ; active : true ; AS1 > ,
  < D1 : dataobject | name : "CI doc" ; status : initial ; AS2 > , A >>
  =>
  CVs * << < X : task | name : "Change CI" ; active : true ; AS1 > ,
  < D1 : dataobject | name : "CI doc" ; status : inUse ; AS2 > , A >> .

(f) Document CI
 rl [DocumentCIp4] :
  CVs * << < X  : task | name : "Document CIs" ; active : true ; AS1 > ,
  < D1 : dataobject | name : "CI doc" ; status : inUse ; AS2 > , A >>
  =>
  CVs * << < X  : task | name : "Document CIs" ; active : true ; AS1 > ,
  < D1 : dataobject | name : "CI doc" ; status : documented ; AS2 > , A >> .

(g) Releasing Baseline
 rl [ReleaseBaseline] :
  CVs * << < X : K | name : "Release Baseline" ; active : true ; hasOutput : D1 ; AS1 > ,  A >>
  =>
  CVs * << < X : K | name : "Release Baseline" ; active : true ; hasOutput : D1 ; AS1 > ,
  CreateDO (d 4, "Baseline", released , X) , A >> .

(h) Baseline made Read Only
 rl [BaselineReadOnly] :
  CVs * << < X : K | name : "Make Baseline Available to Read" ; active : true ; hasOutput : D1 ;
  AS1 > , < D1 : dataobject | name : "Baseline" ; status : released ; AS2 > , A >>
  =>
  CVs * << < X : K | name : "Make Baseline Available to Read" ; active : true ; hasOutput : D1 ;
  AS1 > , < D1 : dataobject | name : "Baseline" ; status : readOnly ; AS2 > , A >> .
```

Figure 3.30: Maude Representation for DSR from (e) to (h) in Figure 3.28

The CI status is then changed to `updated` to indicate changes committed as a result of the activity `Document used CI` being active. The rule (f) in Figure 3.28 and its Maude representation in rule (*Document CI*) in Figure 3.30 simulate this behaviour.

The modifications applied to the configuration item CI are released as a baseline. When the activity `Release Baseline` is active, a data object `Baseline` is created with status (`released`) as shown in rule (g) in Figure 3.28 and Maude representation rule (*Release Baseline*) in Figure 3.30. Finally, a baseline is made a read only document with through the activity *Make baseline available to read*. While this activity is active, the data object `Baseline` status is changed from `released` to `readOnly` as shown in rule

(h) in Figure 3.28 and its Maude representation in rule (*Beaseline made Read Only*) in Figure 3.30.

3.5 Chapter Summary

In this chapter we introduced our formalization in Maude for the syntax and semantics of the core BPMN elements. The proposed formalization considers that the guard expressions are held by the split decision-based gateways and not the outgoing flow attached to them as the BPMN standards specifies. This allows the gateways to be equipped with the condition expressions as well as being able to *decide* on the outgoing flow to consider in the next step without having to actually visit the sequence flows connected to the gateway before the decision is taken. We proposed a context-free grammar for the guard expressions that can be used in evaluating the conditions and hence give the gateway the ability to decide on passing the flow to its children. The notion of well-formed BPs is defined for the formalization for the purpose of validating the structure of the BPMN process models. That is, the well-structured and well-formed BPMN processes are established and the notion of gateways block is introduced.

Using examples with the corresponding formal representation, a set of rewriting rules has been defined in this chapter to formally describe and control the behaviour of a BPMN process. However, we found out that for real life business process models, the execution restrictions are not limited to the syntax and semantics of the modelling tool elements (i.e. BPMN) and it is not enough to represent the behaviour of the basic elements without simulating the specific behaviour requirements for each business context. Hence, we provided a set of domain specific behavioural rules (in Section 3.4.7).

In the next chapter the proposed formalization in this chapter is verified by introducing the soundness analysis and checking for the well-formed BPMN models. Moreover, a simulation analysis with the Petri net mapping for the BPMN is discussed.

Chapter 4

Semantics Verification

Verifying business process models aims at proving that they are free from errors that may lead to unsuccessful execution of the process. Following the fact that solving structural conflicts in testing phases is much more expensive than detecting them in the design phase of business process life cycle [7, 81], the formal verification techniques are eagerly motivated to be applied to the modelling languages.

We discuss soundness property for the proposed semantics verification. The chapter is organized as follows; in Section 4.1, a comprehensive description of the gateway block structure is introduced. This is followed by describing deadlock situations as a structural problem [69] and as a context-related problem in Section 4.2.1. Three new patterns for OR gateways are introduced in Section 4.2.2, followed by relating all the deadlock patterns to the introduced formalization of well-formed BPMN models in Section 4.2.3. Finally, the soundness is formally proved in Section 4.3.

4.1 Gateway Block Structure

The formalization adds a structural restriction in designing BPMN models with gateways to achieve sound business process models. Gateways are designed in a *block*, i.e. each split gateway should have an accompanying merge gateway of the same type. A block has only one entrance point and one exit point. Figure 4.1 presents example AND, XOR, OR blocks where the flow is splitted into two branches to activate activities X, Y or both of them, then the flow is merged at the other side of the block in the corresponding merge gateway.

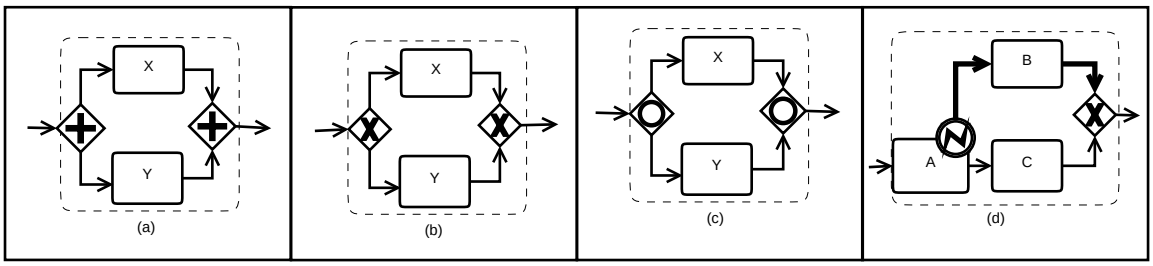


Figure 4.1: Examples of Gateways Block structure.

Exceptions represent a spacial case, where the exception flow (Figure 4.1 (d)) sourced from a boundary-attached error event and the normal flow need to be merged using an XOR merge gateway. In Figure 4.1 (d), the thick arrows represent the exception flows. Notice that XOR merge gateway will expect only one incoming flow in order to get activated, and this matches the fact that either the normal flow or the exception flow only will take place. In this particular case, the XOR merge gateway will refer to the exception event symbol as `itsSplit` attribute value.

4.1.1 AND Block Structure

In case of AND gateway, the flow normally goes from the split gateway towards the merge gateway (c.f. Figure 4.2 (a)), however, in the feedback case, when the merge gateway precedes the split gateway, the model will unsuccessfully terminate. This happens when the AND join gateway tries to merge the input flow and waiting for one of its input flows which comes from the AND split gateway in Figure 4.2 (b), or from activity `X` in Figure 4.2 (c) which result in a deadlock situation.

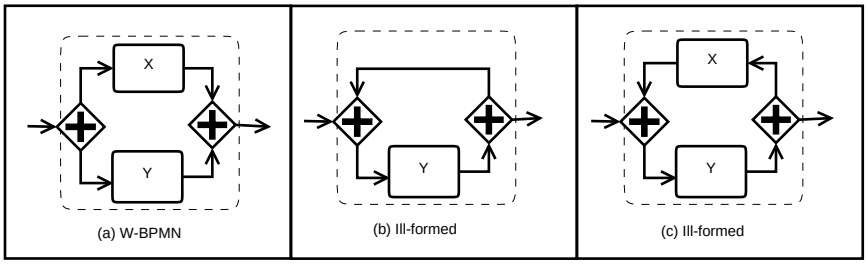


Figure 4.2: Examples of AND gateways Block structure

To avoid this situation in well-formed BPMN models, we add the following condition to the specifications of a well-formed BPMN AND join gateway. The condition states that the AND join gateway should not accept any flow from inside the block it makes. If the AND join gets inputs from inside the block, i.e. a feedback flow, in which case the AND join will be waiting for all its inputs to be activated, and this will never happen in this case. The following is the condition, where functions *inputTrans*, *outputTrans*, *between* are defined in the Maude code files (See Appendix A for details). If the condition above evaluates to *false*, then the AND join gateway is well-formed in the BPMN model, otherwise deadlock situations are possible as illustrated in Figure 4.2 (b) and Figure 4.2 (c).

```
inputTrans(< G1 : ajoingate | in:(tN1,T1); out:tN3; AS1 >, notrans)
in outputTrans(between(< G1:ajoingate | in:(tN1,T1); out:tN3; AS1 >,
                       < G2:aforkgate | out:(tN2,T2); AS2 >,
                       (A,< G2:aforkgate | out:(tN2,T2); AS2 >,
                       < G1:ajoingate | in:(tN1,T1); out:tN3; AS1 >),
                       noobject, noobject), (tN2,T2))
```

Briefly, functions `inputTrans` and `outputTrans` take an object (or a set of objects) and returns all the input and the output transitions for all the objects in the set respectively. Function `inputTrans` is defined below. The operator `in` is a boolean operator which takes two sets of transitions and returns *true* if the first set of transition is *in* the second set of transitions, otherwise it returns *false*.

```
op inputTrans : ObjectSet TransSymbol -> TransSymbol .
eq inputTrans(noobject,T1) = T1 .
eq inputTrans((< X:K | in:T1; AS1 >, A), T2)
 = inputTrans(A,(T1,T2)) [owise] .
```

4.1.2 XOR Block Structure

The situation is a little bit different in case of XOR block. According to the semantics of the XOR merge gateway, it is activated if one of its input flows is coming from an immediate active predecessor object, thus, it does not wait/check other incoming flows. In Figure 4.3, the XOR split and merge gateways in Figure 4.3 (a) and Figure 4.3 (b) follow the block structure, however, Figure 4.3 (a) is a forward representation and Figure 4.3 (b) contains feedback flow. In Figure 4.3 (b), the flow will not deadlock at the XOR merge gateway as it needs only one active predecessor to activate the merge gateway. After that Y is activated then the XOR split gateway is activated. At this point, two cases are possible: (1) if the XOR split guard allows the output flow to leave the loop, then the flow continues normally, or (2) if the XOR split guard allows the feedback flow, then activity X is activated and after that the XOR merge gateway. These two steps are repeated successively and the model enters an infinite loop.

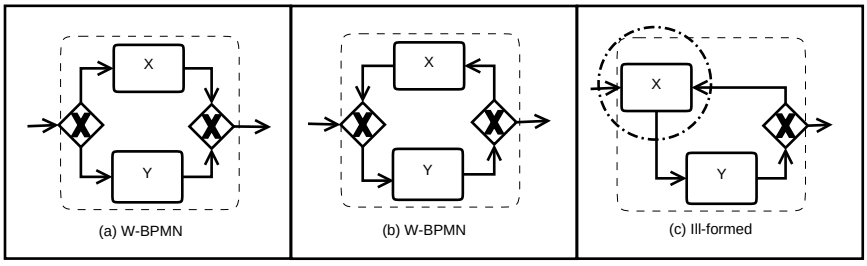


Figure 4.3: Examples of XOR gateways block structure.

In our semantics, the second case is not allowed by introducing domain-specific rules for the model semantics, in which the guard condition has a condition which terminates the loop at certain point. This means that feedback is allowed to happen in the well-formed BPMN models which applies the block structure for gateways. If we do not use the XOR gateways block structure, cases like Figure 4.3 (c) is possible, where as a result of using a feedback flow from an XOR split gateway, an activity X has two input flows, which violates the well-formed BPMN models definition.

4.1.3 OR Block Structure

OR gateways should be represented in a block structure. In Figure 4.4 (a), the OR gateways are in forward positioning; i.e. the split gateway precedes the merge gateway

and the two gateways forms the boundaries for the block. This case is well-formed and does not contain structural errors.

Having feedback in OR gateways is possible in model representations, however, there should be a *limiting* condition that prevent the model from entering an infinite loop (e.g. a vicious circle). Examples of the feedback OR gateway blocks can be found in Figure 4.4 (b) for feedback flow containing an activity and Figure4.4 (c) for feedback flow without activities. If the split gateway in each case has the guard condition allowing only the feedback flow, then an infinite loop is entered unless a stopping point is reached (e.g. a counter is reset) or a new information came available to the OR split gateway to exit the loop. In more details, the OR merge gateway `g1` will wait to know how many incoming flows it should expect to synchronize and get activated. This information comes from its accompanying OR split gateway `g2`. Two examples with three situations are discussed here and illustrated in Figures 4.4 (b) and 4.4 (c).

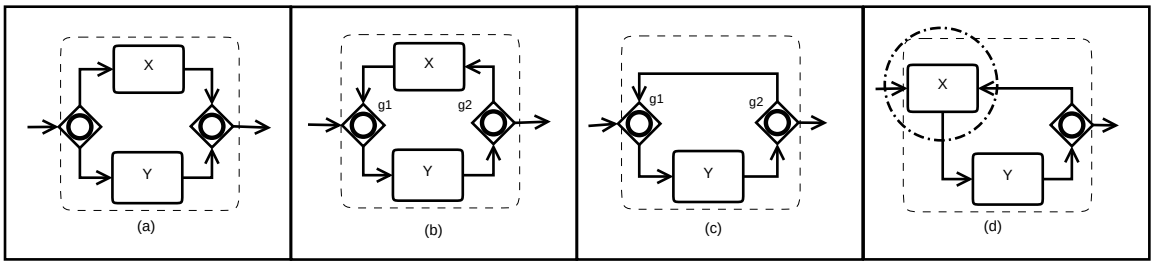


Figure 4.4: Examples of OR gateways block structure.

Case One: in the first attempt, `g1` will be active and then `Y` is activated. This is followed by activating `g2`. If the condition in `g2` allows only the feedback flow that goes as input to `g1`, then `g1` will be active. That is, the feedback flow will be feeding it with the required input flow forever creating an infinite loop.

Case Two: if the condition in `g2` allows only the flow going outside the block, then the `g1` will be active as it gets one input flow from outside the block in the first attempt and activates activity `Y`, then `g2` will be active and the flow proceeds out of the block successfully. In this case activity `X` will not be executed in the model in Figure 4.4 (b). However, there is no necessity to add the OR split and merge gateways in such models, otherwise an infinite loop is entered if the guard condition allows the feedback flow. We discussed this case as it may represent an incorrect implementation for the block structure that may result in an infinite loop.

Case Three: if the condition in g2 allows both the flow going outside the block, and the feedback flow, then the merge gateway g1 knows that it should expect two input flows to synchronize and be activated. In the first execution attempt, g2 will receive only one input from outside the block and will be waiting for the second flow to arrive and this will not happen. A deadlock situation occurs, i.e. the execution of this gateway is repeated and allow for lack of synchronization.

In Figure 4.4 (d), the absence of a merge gateway to close the block results in an activity with more than one input, and this is not allowed in a well-formed BPMN model as discussed before. Moreover, this activity X will need two inputs which will not be available at the same time; i.e. the incoming flow from outside the block to activity X and the outgoing flow from the OR split gateway.

In summary, OR gateways should be presented in a block structure in acyclic models where the feedback flows are not allowed based on the above discussion. One possible solution for the feedback situation is to replace the OR gateways with XOR gateways where only one flow is allowed at the split; i.e. either going outside the block or to feedback with extra guard loop-exit condition . In Chapter 3, OR join gateway semantics introduced a mechanism that links the join gateway with its split gateway in order to keep information of how many activated branches should the join gateway waits for to synchronize. This is still a valid specification for acyclic models.

4.2 Deadlock Patterns

As defined in Definition 3.4.5, a BPMN process execution is represented by an execution path ($\mathcal{EP}$). If the process terminated in a state which is not an end state, then the process is *unsuccessfully* terminated [102, 7]. *Deadlock* is a situation where such behavioural problem occurs in the business process execution paths. A BP model can be designed with these errors unintentionally and hence includes the possibility of being successfully terminated in certain cases due to deadlocks [2]. A BP is in a deadlock situation if a certain state of the model (but not necessarily all) cannot successfully continue its execution, while it has not yet reached the end state. W-BPMN models which do not have gateways cannot suffer from deadlocks. Formally, a deadlock state is the state where an

object that is not an end event is active and it cannot pass the activation to its successors due to unwanted (possibly undefined) behaviour. The state will not find a matching rule left hand side in any of the (defined) semantics behaviour, and therefore the execution will unsuccessfully terminate.

Definition 4.2.1. (Deadlock): The state $S_d \in P(O)$ is a deadlock state iff

(1) it is not an end state; i.e. $S_d \neq S_e$, and

(2) it does not have a successor state ($\forall r \in R(\neg\exists S' \in P(O)(S_d \xrightarrow{r} S'))$).

4.2.1 Structural and Domain-Specific Deadlocks

In the following we will distinguish between two types of deadlocks; structural deadlock and domain specific deadlocks. In the structural deadlock situations, the cause of the deadlock is a structural error caused by the improper use of BPMN elements resulting in errors. It can be a result of divergence and convergence of the flow in the business process (i.e. gateways). For example, if a model contains only activities which are executed sequentially without restrictions or data objects dependencies, then there will not be any possibility for deadlocks. However, if the model contains any splitting or merging behaviour, then the flow is blocked and only being passed through the gateway on successful evaluation of the the guard expressions. Failure to match the split and merge gateways of the same type (i.e. block structures introduced in Section 4.1) may result in gateways waiting for input flows that will not arrive, or gateways where flow is not synchronized. The second type of deadlocks is domain-specific. Since the business process is well-structured and may be well-formed but still suffer from unsuccessful terminations due to objects waiting for a certain resource (i.e. data object) which has not been produced, or lack of exceptions definition for possible semantics errors; e.g. a gateway which all its guard expressions evaluate to *false* and no exceptions are defined as part of its semantics.

In [69], *structural deadlock patterns* are classified into five main patterns based on two concepts; i.e. reachability and absolute transferability. *Reachability* between nodes A and B in a process graph means that there is at least one path from A to B, while *absolute transferability* states that a token (work item in [69]) can always be transferred from

node A to all input points of node B. Thus the absolute transferability reduces reachability between two nodes if there exist routing control nodes (gateways) in between and whenever there is a reachability without absolute transferability, there is a chance for a deadlock [7]. The deadlock patterns according to [69] are explained in terms of AND-split, AND-join, OR-split gateways and start event in Figure 4.5. In order, the patterns are: Figure 4.5(a) work item outflow deadlock type-1, Figure 4.5(b) work item outflow deadlock type-2, Figure 4.5(c) work item deadlock type-3, Figure 4.5(d) loop deadlock type, and Figure 4.5(e), Figure 4.5(f) multiple source deadlock type. The thick lines represents absolute transferability (AT) and the dotted lines represents reachability between two points in the model.

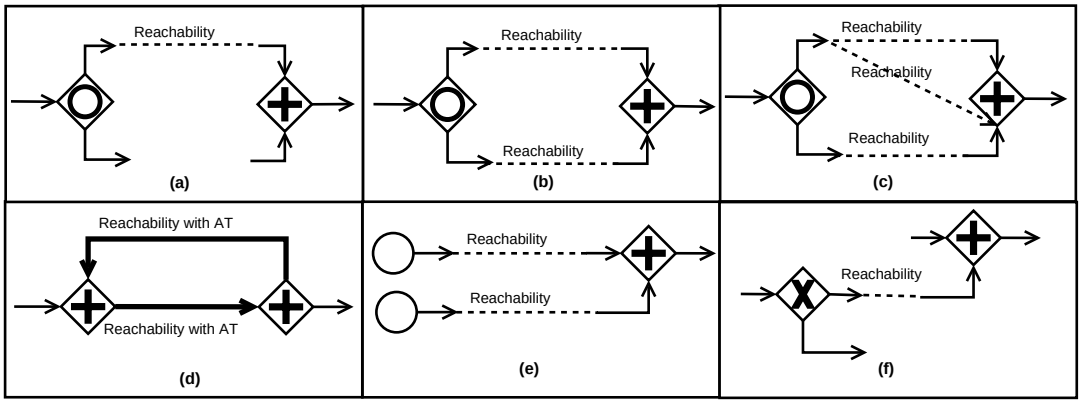


Figure 4.5: Structural deadlock patterns [69]

The first three patterns (a,b,c) in Figure 4.5, represent when the model has an OR-split gateway which one or more of its outgoing flow has a reachability relation to an AND-join gateway. The semantics of the OR-split gateway allow for the activation of one, more, or all of its outgoing flows, while the AND-join semantics enforce the gateway to wait until all its incoming flows reached it, i.e. the join gateway will be waiting for one/more tokens to arrive to it while it is not activated by the OR split gateway from the beginning. This results in a deadlock situation. This is the same case in patterns (a), (b), and (c), no matter how many OR-split outgoing flows are activated, merging them with an AND-join confuses the execution and the process enters a deadlock state (S_d).

In pattern (d), loop deadlock type, AND-join precedes its split AND-split in the model, allowing for a feedback flow to take place. The pattern describes the deadlock situation where the AND-join is reachable from the AND-split through the feedback

flow, while the AND-split is reachable from the AND-join through one of its outflows. The absolute transferability property is held here between the join and fork gateways as the token moves from the join gateway to all the input points of the fork gateway (which happens to be one input flow as it is a fork gateway). However, execution wise, the AND-join gateway will be waiting for the other input flow which will not be active as it is not part of the loop. In this case the deadlock (S_d) occurs at the AND join gateway.

Finally, patterns (e) and (f) assume a model which has multiple sources for the AND-join input flows. This may result because the model includes more than one starting events and one of them is the source for some input flows to the AND-Join (c.f. Figure 4.5 (e)), or one of the AND-join inputs sources from an XOR-split [7] (c.f. Figure 4.5 (f)). In the case of multiple independent start events, it may happen that one of the start events only is activated and then the AND-join gateway will get only one of its input flows and will be waiting for the other input while they will not arrive (as their source event has not been activated), then a deadlock occurs. Similarly, when the input flow for the AND-join is coming from an XOR-split gateway, that means there is a possibility that the XOR-split activates another output flow and leaves the AND-join waiting for it input which results in a deadlock situation.

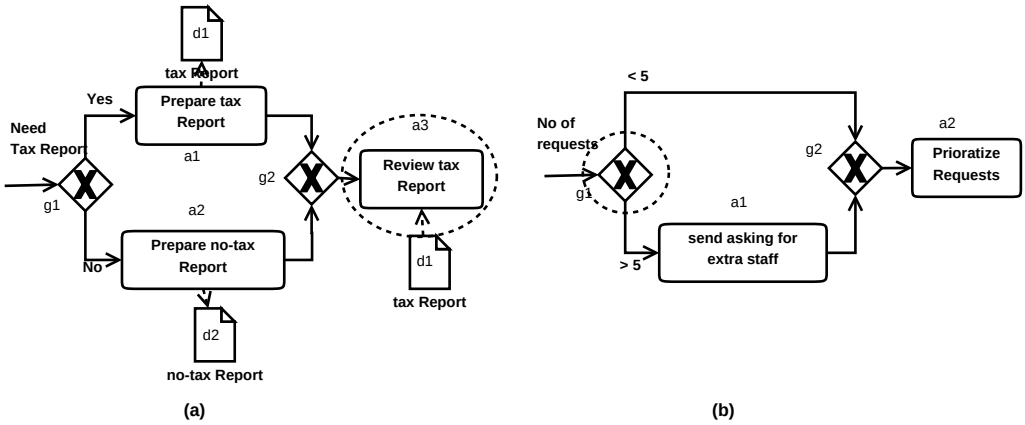


Figure 4.6: Semantics Deadlock Examples

Domain-specific deadlock patterns may occur as a result of not considering some information (e.g. the data objects status change, the guard expressions and their control values) used in the real process in the model design. These conditional restrictions are normally represented in our formalization as domain-specific rules (c.f. Section 3.4.7). In case of having restrictions on the execution of objects, e.g. an activity which has to

wait for a report to be available and this report will never be available (Figure 4.6), then a deadlock can occur. The activity will be waiting in a state without a possible state transformation (i.e. an applicable rewrite rule). For example, in Figure 4.6 (a) an activity `Review tax Report` is waiting for a document `tax Report` which will never be ready. If we represent the process state by the active object(s) it contains and ignoring other inactive objects, we can represent the rewrite steps for the process in Figure 4.6 (a) as follows: $S_1 : (g_1, active) \xrightarrow{[XORsplit]} S_2 : \{(a_2, active), (d_1, initial)\} \xrightarrow{[DSR2]} S_3 : \{(a_2, active), (d_2, created)\} \xrightarrow{[XORmerge]} S_4 : (g_2, active) \xrightarrow{[Seq]} S_5 : \{(a_3, active), (d_1, initial)\} \xrightarrow{?} S_d$ where we represent the object identifier (e.g. `a 1` in Maude mode) as (a_1) in math mode.

The application of rule [*Seq*] (introduced in Section 3.4.2) is represented as:
$\{(X, active), (Y, inactive)\} \xrightarrow{[Seq]} \{(X, inactive), (Y, active)\}$
with the left-hand side term as the set of states of the objects participating in the rule, and the right-hand side term as the set of the resulting objects states. The rules names over the rewrite arrows are the semantic rules labels from Chapter 3 and the domain-specific rules (i.e. [*DSR1*], [*DSR2*], and [*DSR3*]) are interpreted as below:
$\{(a_1, active), (d_1, initial)\} \xrightarrow{[DSR1]} \{(a_1, active), (d_1, created)\}$,
$\{(a_3, active), (d_1, created)\} \xrightarrow{[DSR3]} \{(a_3, active), (d_1, reviewed)\}$,
$\{(a_2, active), (d_2, initial)\} \xrightarrow{[DSR2]} \{(a_2, active), (d_2, created)\}$.

Another possible deadlock situation occurs when a gateway guard has fed with a value which is not defined in the designed expressions. For example, in Figure 4.6 (b), the XOR gateway guard checks for the number of certain requests (`No. of requests`), if the number is less than five, then the requests will be prioritized, or if they are greater than five, the responsible is asked to `send for extra staff` to assist, then they can `prioritize requests`. The deadlock happens if the number of requests is exactly five, in which case, there is no defined processing for such value and the process terminates unsuccessfully ($S_1 : (g_1, active) \xrightarrow{[XORsplit]} S_d$). The BPMN semantics defined in [68] defined an exception to be attached to each split gateway in case that the guard condition did not evaluate to a defined value, i.e. all the guard conditions failed to evaluate to *true*. This type of exceptions is essential to prevent such deadlocks in BPMN models. In section 3.4.2 and Section 3.4.2, this type of exceptions with the gateway semantics has

been defined.

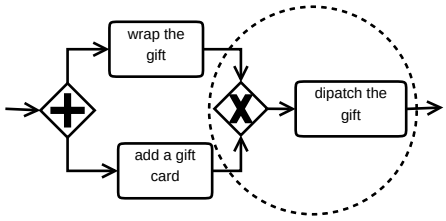


Figure 4.7: Lack of Synchronization Example

A related BPMN control flow error is the lack of synchronization. This happens as a result of having a structural error in the BPMN model which allows for an object to be activated (executed) more than once without design purpose, i.e. an XOR merge gateway that merges the flow splitted by an AND fork. For example, in Figure 4.7, the parallel activities `wrap the gift` and `add a gift card` will be executed concurrently, however, the first to finish (assume it is activity `wrap the gift`) will activate the XOR merge gate and then activate the activity `dispatch the gift` as the XOR merge needs one active predecessor to be activated. At the same time, when the other activity (`add a gift card`) is completed, it activates the XOR merge gateway again and then activates the activity `dispatch the gift`. This means the gift will be dispatched twice, which is not a desired procedure for the business process. Technically, the synchronization of the parallel activities failed at the XOR merge gateway.

4.2.2 More Deadlock Patterns with OR gateways

During our work for formalizing the BPMN models and the above discussion about the block structure for the gateways, we found out that there are other patterns which can cause deadlock to occur in the models including OR gateways. In this section we will focus on the deadlock situations which the OR gateway may be involved in. Figure 4.8 illustrates these patterns graphically. The first pattern (c.f. Figure 4.8 (a)) describes the situation where the activity `X` will need two input flows in order to be active. While this is not allowed as a well-formed BPMN model, having the OR gateway may seems to make it a possible implementation, however, it is not as discussed above in Figure 4.4 (d).

The second deadlock pattern in Figure 4.8 (b) represent the feedback situation in an OR gateway block, which has been already explained in Figure 4.4 (b). According to the

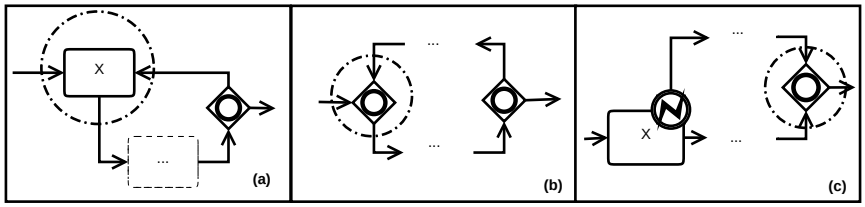


Figure 4.8: More Deadlock Patterns

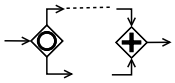
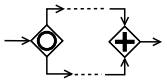
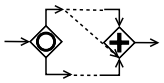
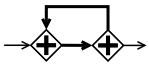
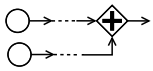
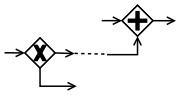
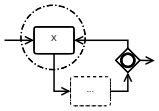
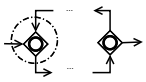
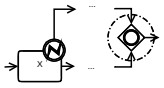
OR merge gateway semantics, the gateway is activated when it receives the same number of activated input flows as its corresponding OR split gateway activated. In case that the two output flows are activated by the split gateway, that means the merge gateway will wait for two incoming flows while it will receive only one from outside the block and unsuccessfully terminates.

The third deadlock pattern in Figure 4.8 (c) in case the exception flow is merged with the normal flow using an OR merge gateway. Again the specified semantics for the OR merge gateways assumes the block structure, and the split gateway is preceding the merge gateway in execution. In this case, the process will deadlock as the OR merge gateway will be waiting for the information from its corresponding OR split gateway, which is missing already from the model.

4.2.3 Relating to The Proposed Formalization

In our proposed formalization, we tried to restrict the resulting models to avoid the deadlock patterns discussed above. Table 4.1 summarizes the deadlock patterns and relate them to significant parts of our formalization. Refer to Definition 3.3.2 and Section 4.1 for more details on the gateways block structure as a syntactic condition for well-formed BPMN models. Pattern-5 is for multiple sources deadlock type, therefore, a well-formed BPMN model has only one start event, which avoids the first case in this pattern. Moreover, the gateways are required to be represented in a block structure, i.e. each split gateway has a companion merge gateway of the same type. This restriction prevents the deadlock states in Pattern-1, Pattern-2, Pattern-3, and Pattern-6. The condition explained in Section 4.1 for AND join gateways, that it only accepts input flows from outside the block is aiming at avoiding Pattern-4. For Pattern-7, the well-formed BPMN models definition restricts the activities to have only one incoming flow and one outgoing

Table 4.1: Deadlock Patterns and The Proposed Formalization

Pattern No.	**Figure**	**Syntactic Condition**
Pattern-1		Block structure (same type gateways)
Pattern-2		Block structure (same type gateways)
Pattern-3		Block structure (same type gateways)
Pattern-4		Block structure (AND feedback condition)
Pattern-5		W-BPMN models, Block structure
Pattern-6		Block structure (same type gateways)
Pattern-7		W-BPMN models, Block structure
Pattern-8		Block structure, Domain-specific Rules
Pattern-9		W-BPMN models, Block structure

flow, which will avoid such pattern. Pattern-8 is avoidable with if the modeller defines domain-specific rules that manage the flow in the feedback situations. Finally, Pattern-9 is avoidable through the condition in the well-formed BPMN models that exception flow is merged with the normal flow using only an XOR merge gateway.

In the next section, we are formally proving the soundness of the well-formed BPMN models based on the classical soundness definition in [92, 102].

4.3 Soundness

Soundness of workflow models has been proposed as a correctness criterion verifying different BP formalizations (e.g. [92, 55, 102]). However, there are many different notions of soundness in literature (the interested reader can refer to [95] for a comprehensive discussion on workflow soundness and its decidability). Discussing the Petri net's formalization for workflow models in [92], the authors defined the soundness as a correctness criterion for the resulting workflow models. This definition was referred to as the *classical soundness* later in [95], where the authors distinguished among six different notions of soundness in literature. Generally, a business process model is sound if it can successfully terminate without left over active objects and all the model objects can be activated in one of the execution traces. The model should be free from errors, e.g. deadlocks, which cause unplanned termination of the execution.

In [81, 93, 100], the sound model is free from control flow errors. The soundness can be defined as freedom of deadlock and lack of synchronization. As we explained, informally, in Table 4.1 in Section 4.2.3, the proposed well-formed BPMN model can guarantee deadlock freedom and provide mechanisms to avoid lack of structural synchronization.

We are going to use a definition similar to the definition in [92, 102] for classical soundness to prove that well-formed BPMN models are sound. First, we update our definition of reachability; a state state S_2 is *reachable* from state S_1 if there is an execution path from S_1 to S_2 (i.e. $S_1 \xrightarrow{*} S_2$)[1].

[1]We use the symbol $\xrightarrow{*}$ to denote a sequence of zero or more rewrite steps, the symbol $\xrightarrow{+}$ to denote a sequence of one or more rewrite steps and the symbol $\xrightarrow{r}$ to specify the rewrite rule r is applied in this step.

Definition 4.3.1. (Sound): A W-BPMN model is sound iff:

(i) [***option to complete***] For every state S reachable from the start state S_s, there exist an execution path leading from S to the end state S_e; i.e. $\forall S \in P(O)(S_s \xrightarrow{+} S \xrightarrow{+} S_e)$,

(ii) [***proper completion***] If the end event is active, then all other objects are inactive in the same process state (i.e. process in state S_e); i.e.
$(\exists(o_{id}, s_{fo}) \in S(o_{id} = ObjId(o) \wedge ObjCid(o) = \texttt{endEvent} \wedge s_{fo} = active) \wedge$
$\neg\exists(o'_{id}, s'_{fo}) \in S(o'_{id} = ObjId(o') \wedge s'_{fo} \in \{active, ready2bActive\}) \wedge o \neq o')$
$\Rightarrow S = S_e$,

(iii) [***no dead objects***] There are no dead objects in the model, i.e. it should be possible to execute an arbitrary object in the model in one or more of the traces; i.e. $\forall(o_{id}, s_{fo}) \in S(\exists(o'_{id}, s'_{fo}) \in S'(S \xrightarrow{*}_{\mathcal{R}} S' \wedge s_{fo} \neq s'_{fo})$ and $\forall(o_{id}, s_{do}) \in S(\exists(o'_{id}, s'_{do}) \in S'(S \xrightarrow{*}_{\mathcal{R}} S' \wedge s_{do} \neq s'_{do})$.

For proving the soundness of the well-formed BPMN models, we will define two models. The first is the behavioural W-BPMN model which is defined by adding the set of rewrite rules R representing its behaviour and the set of process states $P(O)$. The second one is an extended behavioural well-formed model to extend the behavioural W-BPMN model with a feedback sequence flow t_x connects the end object and the start object and a rewrite rule r_x that re-activates the process after it is completed. The set of states for the extended model is the same for the original model as no new states are added with the extension (i.e. $\overline{P(O)} = P(O)$).

Definition 4.3.2. (W-BPMN$_b$): A well-formed W-BPMN behavioural model (W-BPMN$_b$) is a triple $(O, R, P(O))$ where $O = (OS, T)$ is a well-formed BPMN model that contains BPMN objects OS and the set of their connecting flow transitions T, $R = \{r_1, \ldots, r_m\}$ is a finite set of rewrite rules defined in rewrite theory $\mathcal{R}$, and $P(O)$ is the set of process states for the model.

Definition 4.3.3. ($\overline{\text{W-BPMN}_b}$): Let W-BPMN$_b$ = $(O, R, P(O))$ be a well-formed behavioural model. An extended well-formed behavioural BPMN model ($\overline{\text{W-BPMN}_b}$) is defined as a triple $(\overline{O}, \overline{R}, \overline{P(O)})$ where $\overline{O} = (OS, T \cup \{t_x\})$, $\overline{R} = R \cup \{r_x\}$, and $\overline{P(O)} = P(O)$. The sequence flow t_x is added to link the end object (o_e) with the start object (o_s) and used

in the soundness proof and rule r_x is defined below to deactivate the end object and reactivate the start object.

```
rl [rx] :
   << < X : endEvent | active : true ; AS1 > ,
      < Y : startEvent | active : false ; AS2 >,A >>
=> << < X : endEvent | active : false ; AS1 > ,
      < Y : startEvent | active : true ; AS2 >,A >> .
```

We define two properties for the well-formed models which are necessary for the proof; i.e. live and path-complete properties.

Definition 4.3.4. (Live Property): A $\texttt{W-BPMN}_b$ model is live iff, for every state S, there is a state S' which is reachable in one rewrite step: $\forall S \in P(O)(\exists S' \in P(O)(\exists r \in R(S \xrightarrow{r} S')))$.

Definition 4.3.5. (Complete-Path Property): A $\texttt{W-BPMN}_b$ model is path-complete iff it has at least one complete execution path $c\mathcal{EP}$[2].

Given that the state space is the same for the models $\texttt{W-BPMN}_b$ and $\overline{\texttt{W-BPMN}_b}$, therefore, the extended execution path $\overline{c\mathcal{EP}}$ can be defined as $\overline{c\mathcal{EP}} = c\mathcal{EP} \cup \{(S_e \xrightarrow{r_x} S_s)\}$.

For arbitrary well-formed BPMN model and the corresponding extended model, we prove: $\texttt{W-BPMN}_b$ is sound if and only if $\overline{\texttt{W-BPMN}_b}$ is live and path-complete.

First, we prove the if direction,

Lemma 1. If $\overline{\texttt{W-BPMN}_b}$ is live and path-complete, then $\texttt{W-BPMN}_b$ is sound.

Proof. Let $S_s, S_e \in \overline{P(O)}$, where S_s is a start state and S_e is an end state.
S_s can be rewritten into a state S_1 in one rewrite step, and S_e can be a result of rewriting a state S_n in one rewrite step:
$\therefore S_s \xrightarrow{r_1} S_1$ and $S_n \xrightarrow{r_m} S_e$.
$\because \overline{\texttt{W-BPMN}_b}$ is path-complete
$\therefore \exists es, es' \in \overline{\mathcal{EP}}(source(es) = S_s \wedge target(es') = S_e))$.
which means (S_s, r_1, S_1) and $(S_n, r_m, S_e) \in \overline{\mathcal{EP}}$.
$\because \overline{\texttt{W-BPMN}_b}$ is live

[2] Refer to Definition 3.4.6

$\therefore \forall S \in \overline{P(O)}(\exists S' \in \overline{P(O)}(\exists r \in \overline{R}(S \xrightarrow{r} S')))$.

$\therefore$ for an arbitrary state S (i.e. $S \notin \{S_s, S_e\}$), we have $(S_k, r_k, S) \in \overline{\mathcal{EP}}$. (Definitions 3.4.5).

$\therefore S$ is reachable from S_s (i.e. $S_s \xrightarrow{+} S$).

Applying the same induction step for S and S_e, we get S_e is reachable from S (i.e. $S \xrightarrow{+} S_e$).

$\because \overline{P(O)} = P(O)$

$\therefore \forall S \in P(O)(S_s \xrightarrow{+} S \xrightarrow{+} S_e)$ — [Req(i)]

$\because \overline{\texttt{W-BPMN}_b}$ is path-complete

$\therefore \exists es, es' \in \overline{\mathcal{EP}}(source(es) = S_s \wedge target(es') = S_e))$.

$\therefore S_e \in \overline{P(O)}$.

$\because \overline{P(O)} = P(O)$.

$\therefore S_e \in P(O)$.

$\because \overline{c\mathcal{EP}} = c\mathcal{EP} \cup \{(S_e, r_x, S_s)\}$ by definition,

from [Req(i)]: $S_s \xrightarrow{+} S_e$, from end state (S_e) definition (Definition 3.4.3),

$\therefore (\exists (o_{id}, s_{fo}) \in S_{OS}(o_{id} = ObjId(o) \wedge ObjCid(o) = \texttt{endEvent} \wedge s_{fo} = active) \wedge$

$\neg \exists (o'_{id}, s'_{fo}) \in S_{OS}(o'_{id} = ObjId(o') \wedge s'_{fo} \in \{active, ready2bActive\}) \wedge o \neq o') \Rightarrow S_{OS} = S_e$. — [Req(ii)]

$\because \overline{\texttt{W-BPMN}_b}$ is live

$\therefore \forall S \in \overline{P(O)}(\exists S' \in \overline{P(O)}(\exists r \in \overline{R}(S \xrightarrow{r} S'))), \overline{R} = r \cup \{r_x\}$

$\because R$ contains two types of the possible rewrite rules for a model $\overline{\texttt{W-BPMN}_b}$; one set of rules contains the general behaviour rules for flow object (FO) (discussed in Section 3.4.2) and the other contains the domain-specific rules basically for data objects (DO) (discussed in Section 3.4.7).

$\therefore \forall (o_{id}, s_{fo}) \in S(\exists (o'_{id}, s'_{fo}) \in S'(S \xrightarrow{*}_{\mathcal{R}} S' \wedge s_{fo} \neq s'_{fo})$ and $\forall (o_{id}, s_{do}) \in S(\exists (o'_{id}, s'_{do}) \in S'(S \xrightarrow{*}_{\mathcal{R}} S' \wedge s_{do} \neq s'_{do})$. — [Req(iii)]

From Req(i), Req(ii), Req(iii) and the Soundness definition (Definition 4.3.1),

$\therefore \texttt{W-BPMN}_b$ is a sound BPMN model. □

Lemma 2. If $\texttt{W-BPMN}_b$ is sound, then $\overline{\texttt{W-BPMN}_b}$ is path-complete.

Proof. $\because \texttt{W-BPMN}_b$ is sound

$\therefore \forall S \in P(O)(S_s \xrightarrow{+} S \xrightarrow{+} S_e)$. — [Req(i)]

if $\texttt{W-BPMN}_b$ is not path-complete, then

$\neg\exists es, es' \in \mathcal{EP}(source(es) = S_s \wedge target(es') = S_e)$.

this contradicts the assumption and Req(i) in Definition 4.3.1, which assumes that every state reachable from S_s belongs to an execution path to S_e.

$\therefore$ it is not possible that $\texttt{W-BPMN}_b$ is sound and not path-complete at the same time,

So, if $\texttt{W-BPMN}_b$ is sound, then it is path-complete.

But $\overline{c\mathcal{EP}} = c\mathcal{EP} \cup \{(S_e, r_x, S_s)\}$ by definition.

$\exists \overline{c\mathcal{EP}} \in O_{Paths}(\exists es, es' \in \overline{c\mathcal{EP}}(source(es) = S_s \wedge target(es') = S_e))$.

$\therefore \overline{\texttt{W-BPMN}_b}$ is path-complete.

$\therefore$ if $\texttt{W-BPMN}_b$ is sound, then $\overline{\texttt{W-BPMN}_b}$ is path-complete. □

Lemma 3. If $\texttt{W-BPMN}_b$ is sound, then $\overline{\texttt{W-BPMN}_b}$ is live.

Proof. $\because \texttt{W-BPMN}_b$ is sound

$\therefore \overline{\texttt{W-BPMN}_b}$ is path-complete — [Lemma 2]

$\therefore \exists es, es' \in c\mathcal{EP}(source(es) = S_s \wedge target(es') = S_e)$.

$\because \texttt{W-BPMN}_b$ is a well-formed BPMN model by definition

$\therefore \exists S_s, S_e \in P(O)$.

$\because \overline{P(O)} = P(O)$.

$\therefore \exists S_s, S_e \in \overline{P(O)}$.

$\because \texttt{W-BPMN}_b$ is sound

$\therefore \forall S \in P(O)(S_s \xrightarrow{+} S \xrightarrow{+} S_e)$, — [Req(i) in Definition 4.3.1]

$\because \overline{c\mathcal{EP}} = c\mathcal{EP} \cup \{(S_e, r_x, S_s)\}$ by definition

$\therefore \forall S \in \overline{P(O)}(\exists S' \in \overline{P(O)}(\exists r \in R(S \xrightarrow{r} S')))$

$\therefore \overline{\texttt{W-BPMN}_b}$ is live — [Definition 4.3.4]. □

Theorem 1. A behavioural well-formed BPMN model ($\texttt{W-BPMN}_b$) is sound if and only if ($\overline{\texttt{W-BPMN}_b}$) is live and path-complete.

Proof. It follows directly from Lemma 1 (the if direction), and from Lemma 2 and Lemma 3 (the only if direction)

$\therefore \texttt{W-BPMN}_b$ is sound if and only if $\overline{\texttt{W-BPMN}_b}$ is live and path-complete. □

We have formally proved that the well-formed BPMN models are sound based on the classical soundness definition for workflow models [92]; i.e the well-formed BPMN models are deadlock-free and do not suffer from lack of synchronization.

4.4 Chapter Summary

In this chapter we discussed the structural and model-specific errors that leads to deadlock situations and hence unsuccessful termination of BPs models. A comprehensive view on the BPMN models possible structural and semantic errors which may lead to unsuccessful termination of process through the problems of deadlock and lack of synchronization was introduced. A relevant set of deadlock situations in which OR gateways are involved are discussed.

Where the soundness can be defined as the absence of deadlocks and lack of synchronization, the proposed semantics showed, informally, that well-formed BPMN models are sound. While following the classical soundness definition, we formally proved that well-formed BPMN models (W-BPMN) are sound. These two results allowed us to introduce the application side of the semantics in the next chapter. A CMMI compliance checking problem is introduced and we propose a solution for it using model checking.

Chapter 5

Business Processes Compliance Checking

The problem of checking BPs compliance with certain standard models or requirements as a model checking problem (Section 5.1) has three main challenges: first, the BP modelling language discussed in the previous chapter, second, the formal mapping of requirements into property specifications (Section 5.2), and third, the checking procedure. In this chapter, we provide the details of the CMMI-CM2LTL procedure and Model Checking procedure illustrated in Figure 1.1 where the well-formed BPMN models are checked against the formally represented CMMI-CM requirements. The requirements are mapped into LTL properties through the use of compliance patterns [33] (i.e. inspecting the second challenge) in Section 5.2.3. Based on the fact that Maude has its own LTL model checker and the reasons of choosing model checking technique mentioned in Chapter 1, we use the Maude LTL model checker as the compliance checking procedure (i.e. tackling the third challenge). In Subsection 5.1.2 two examples are introduced; (1) `Release Baseline` example introduced in Chapter 3 and (2) `EX2`, a CM process based on IBM CCM Process (Tivoli) [3]. Finally, the model checking results are discussed in Section 5.3.

5.1 Compliance Checking as a Model Checking

Business process compliance checking has been in the focus for more than two decades alongside with the rise of software quality assurance and process improvement models (e.g. [64, 80, 49, 32, 6]). The urgency of studying such verification approaches is twofold; first, it helps the company to grow based on a solid infrastructure BP which is flexible enough to cope with the improvements over time, second, it is shown that businesses whose BPs are not compliant with some standards, regulations or quality measures may experience some level of failure (e.g. Enron scandal) [32]. As discussed in Section 2.3.2 and shown in Table 2.2, the CMMI-based compliance checking approaches (i.e. appraisals) can be expensive in terms of all kinds of cost (i.e. time, effort, labour, and money). This is due to the checking being dependent on finding the evidences in the SME that proves that it is following certain practices via locating documents and interviewing employees to reach the affirmations and artifacts proving these aspects. Having a well-designed BP will allow the appraisal team to step forward and start from the actual implementation of the process and will not spend the time reviewing the designed processes. The output of the compliance checking is supposed to give the formal evidence for a business process that it obeys the legal, safety, organizational and/or technical requirements of the reference standards and regulations. It shows how the company is flexible and adaptable in the market in a way that guarantees its sustainability.

Starting from a designed BP, and a set of textual compliance requirements, the problems becomes to check if the BP is compliant with (i.e. *satisfies*) these requirements. As we already have formalized the well-formed BPMN models in Chapter 3, then the requirements should be formally represented into formal properties, and hence could be checked. This structure of verification approaches is more applicable with model checking (refer to model checking explanation in Section 2.2.3). Therefore, this compliance checking problem can be dealt with as a model checking problem (e.g. [49, 32, 6]) where the model checking technique can be used to automate the production of such formal evidences based on the input BP model and properties.

Having the fact that model checkers suffer from some limitations, the majority of them are not likely to affect the validity of the proposed approach. In the following points we discuss the known model checking limitations and if they are affecting our

approach or not.

1. Model checking does not provide correctness proofs, unlike the theorem proving techniques [9, 46]. The application of model checking on hand does not require automatic theorem proving. The core task is to check if a BP model satisfies a property.

2. Completeness of the checked system is not guaranteed [9, 46], as the properties being checked can only be decided. However, other unchecked properties cannot be judged. Therefore, the proposed compliance approach provide the mechanism to choose the relevant properties which are in a process area to check. This produces convenient results to the compliance checking problem with the need to know only the satisfaction status.

3. In case of infinite-state systems, abstraction is needed to be applied to the original system to get a relatively finite-state model to be checked. There is still a possibility that the abstraction process is not accurate enough to represent the original model [9, 46, 20]. Moreover, abstractions needs experts to perform them,

4. State-explosion problem [19] forms an issue for model checking big models. In some models, when the number of the states increases, the complexity of the verification procedure used in the model checker increases [9, 46] making it impossible to be operated with computer memory [19]. There are methods that have been developed to overcome this problem (e.g. [71, 36]), however, realistic models, possibly BP models, may still suffer from it. There is still the fact that a SME's BP is usually divided into (possibly cooperating) small processes which model different aspect of work being conducted in the SME to produce a product or a service. Moreover, the compliance requirements are classified into process areas to facilitate the focus, applicability and organization of the related requirements for a specific BP area. Therefore, on checking the BP(s) compliance with requirements, we do not expect models to suffer from state space explosion.

5.1.1 Predicates

The set of predicates Π is used for defining the queries about the model (i.e. representing the properties). Here we present the predicates that we used in the properties afterwards.

- **`executed`**: for the active state of an activity (i.e. $executed(name(o)) = true$ iff $\exists(o_{id}, s_{fo}) \in S(o_{id} = ObjId(o) \wedge o \in FO)$. The function ($name : Oid \rightarrow String$) is defined to take an object identifier and returns its name. The Maude definition for the function specifies that the function takes an object name and retrieves *true* in case it is executed in one of the process states.

```
op executed : String -> Prop .
eq (CVcol * << < X : K | name : PN1 ; active : true ; AS1 >, A >>)
|= executed (PN1) = true .
```

- **`status`**: for the data object status. Data objects change their status during the process execution, and sometimes the status change can be an evidence of certain execution steps (i.e. $status(name(ObjId(o)), s_{do}) = true$ iff $\exists(o_{id}, s_{do}) \in S(o_{id} = ObjId(o) \wedge o \in DO)$. In the Maude definition below, the operator `status` takes a data object name and status, then returns *true* if the state space contains a state in which this data object has this status (i.e. `DOS`).

```
op status : String DOstatus -> Prop .
eq (< d N1 : DT | name : S1 ; status : DOS ; AS1 >, A)
 |= status (S1, DOS) = true .
```

- **`conditionGuard`**: for a control value which is used in the process execution (i.e. if a certain decision is made during the process execution). The operator `conditionGuard` takes the control value and returns *true* if a split gateway has it as its assigned control value (i.e. the corresponding guard expression evaluated to true). Let *cv* be a control value for some decision gateway object $o \in G$, then $conditionGuard(cv) = true$ iff $\exists(o_{id}, s_{fo}) \in S(\exists o \in G(o_{id} = ObjId(o) \wedge s_{fo} = active \wedge cv \in controlValues(o))$. The function ($controlValues : Object \rightarrow ControlValue$) takes the gateway object and retrieves its assigned control values.

```
op conditionGuard : ControlValue -> Prop .
eq << < G1 : xsplitgate | controlValues : (CVs1 .. CVs2) ; AS1 >, A >>
 |= conditionGuard(CVs1) = true .
eq << < G1 : osplitgate | controlValues : (CVs1 .. CVs2) ; AS1 >, A >>
 |= conditionGuard(CVs1) = true .
```

- The **`false case`** for the predicates propositions is defined as the *otherwise* equation.

```
eq TO |= PR = false [owise] .
```

For example if the property requires that an object with name : "Release Baselines") is being executed in a certain state, then the unary predicate `executed` will be used as follows: `executed("Release Baselines")`. As a binary predicate example, we use `status`, which takes the data object name and status. For the data object *CR* with a status *open*, the predicate should be: `status ("CR", open)`. We use the introduced predicates in formalizing the properties in LTL as will discussed below. This is to demonstrate the applicability of the proposed business process formalization in checking their compliance with standard process improvement requirement (e.g. CMMI).

The state predicates are typically not related to the semantics of the business process models, however, they represent certain states of the system specified by the model *M* and they are used to specify some properties [20], hence they are part of the property specification. For this purpose, the module `M-PREDS` is designed to *protect* the well-formed BPMN semantics (in module `BPMN-SEMANTICS`) and to *include* the Maude's predefined `SATISFACTION` module. The later defines the satisfaction relation semantics for a model and LTL properties.

```
mod M-PREDS is
  protecting BPMN-SEMANTICS .
  including SATISFACTION . ...
```

The module `SATISFACTION`, below, defines the satisfaction relations (operator_`|=`_), where `state` and `Prop` are unspecified sorts and `Bool` is for the sort of boolean values which will hold the answer from the property checking. The operator has the attribute

`frozen` which indicate that are no sub-terms included into the term can be evaluated using this operator.

```
fmod SATISFACTION is
  protecting BOOL .
  sorts State Prop .
  op _|=_ : State Prop -> Bool [frozen] .
endfm
```

In Maude's notation, the model checker uses the following command to check if the property is satisfied in the model.

`reduce in` ⟨*ModuleName*⟩ : (`modelCheck`⟨*InitialState*⟩, ⟨*LTLproperty*⟩)

The command `modelCheck` takes the initial state of the system (`initial`), where the process is *inactive* and the LTL formula for the property. The model checker is then perform an extensive state space search for the negation of the property. If a matching for the negation is found, then it returns one trace of states which starts by the initial state and ends with the state where the property is violated (i.e. a counterexample). Otherwise, it returns *true* indicating that the available finite state space does not include any state which violates this property.

5.1.2 BPs Model Examples (System Specifications)

Well-formed BPMN models introduced in Chapter 3 represent the system specifications for the compliance problem we address. Recall the example "Release Baselines" which was introduced in Chapter 3 with Figure 5.1. The model represents a sub-process of the CM process in a software company. It acts as the model specifications specified by the rewrite theory $\mathcal{R}$ earlier in Chapter 3, while the property specifications φ are modelled from the CMMI best practices [22] as explained later in Section 5.2.

Another example of the CM process (EX2) is the one represented in Figure 5.2, which is our interpretation for the IBM Change and Configuration Management in Tivoli [3]. We used the IBM online materials[1] for the IBM CCM process included in the Tivoli tool to produce the BPMN example below. The Change and Configuration Management process contains two sub-processes: `Update CIs` sub-process and `Audit CIs`

[1] Documentation can be found in: [3]

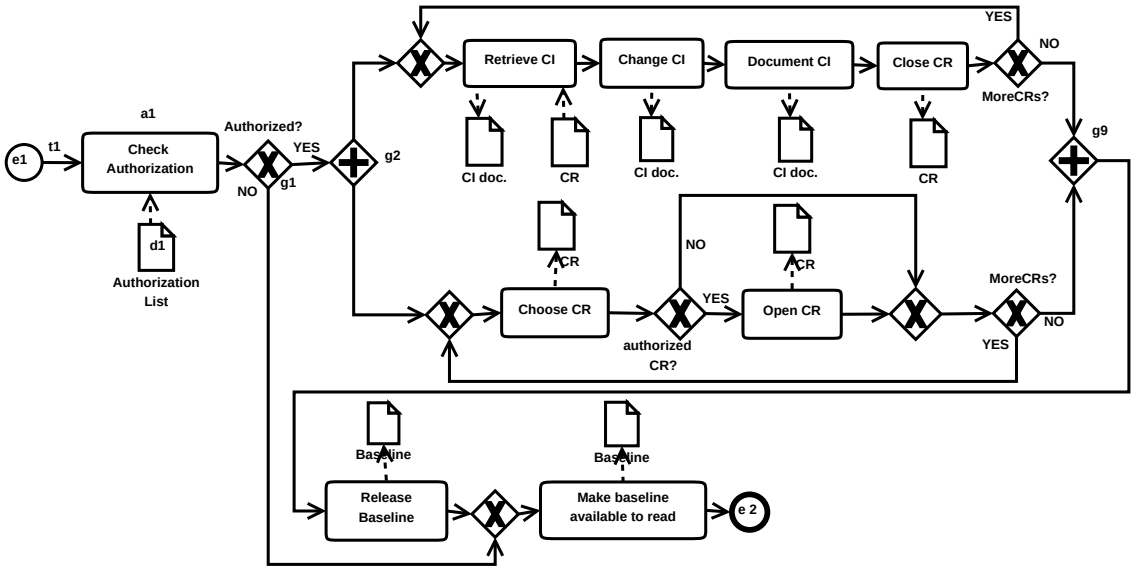


Figure 5.1: EX1 : Release Baseline Model

sub-process. The first starts with validating a received change request `CR`. If it is valid, the `CR` is open and the included changes are prioritized, the relative `CI doc` is retrieved and updated, then the `CR` is closed. Otherwise, if the `CR` is not valid, then it is simply closed. In the second sub-process, an `Audit` document is open, the stakeholders are informed of either the change or invalidity of the request. Upon this, the audit is investigated for variances (e.g. inconsistencies). On acting on variances, the relative `CI doc` is retrieved and the variances are specified. If the validated values are consistent with the actual values, then another check is conducted if the configuration item is protected then a problem in audit is reported, or if they are not protected, then the changes in the `CI doc` are documented. In both cases, a report `Act-on-variance Report` is created. After that the audit document is reported and closed. Finally, in the main process, the `Baseline` document is released.

The example EX2 is modelled in BPMN in Figure 5.2. The data objects status changes are as follows: change request document (`CR: initial -> open -> closed`), the audit document which is created at run time (`Audit: open -> closed`), the baseline document which is created at run time (`Baseline: initial -> released`), the configuration item document (`CI doc: initial -> inUse -> updated -> documented`), the variance handling document which is created at run time (`Act-on-Variance Report: created`). The full Maude representation and the corresponding domain specific rules

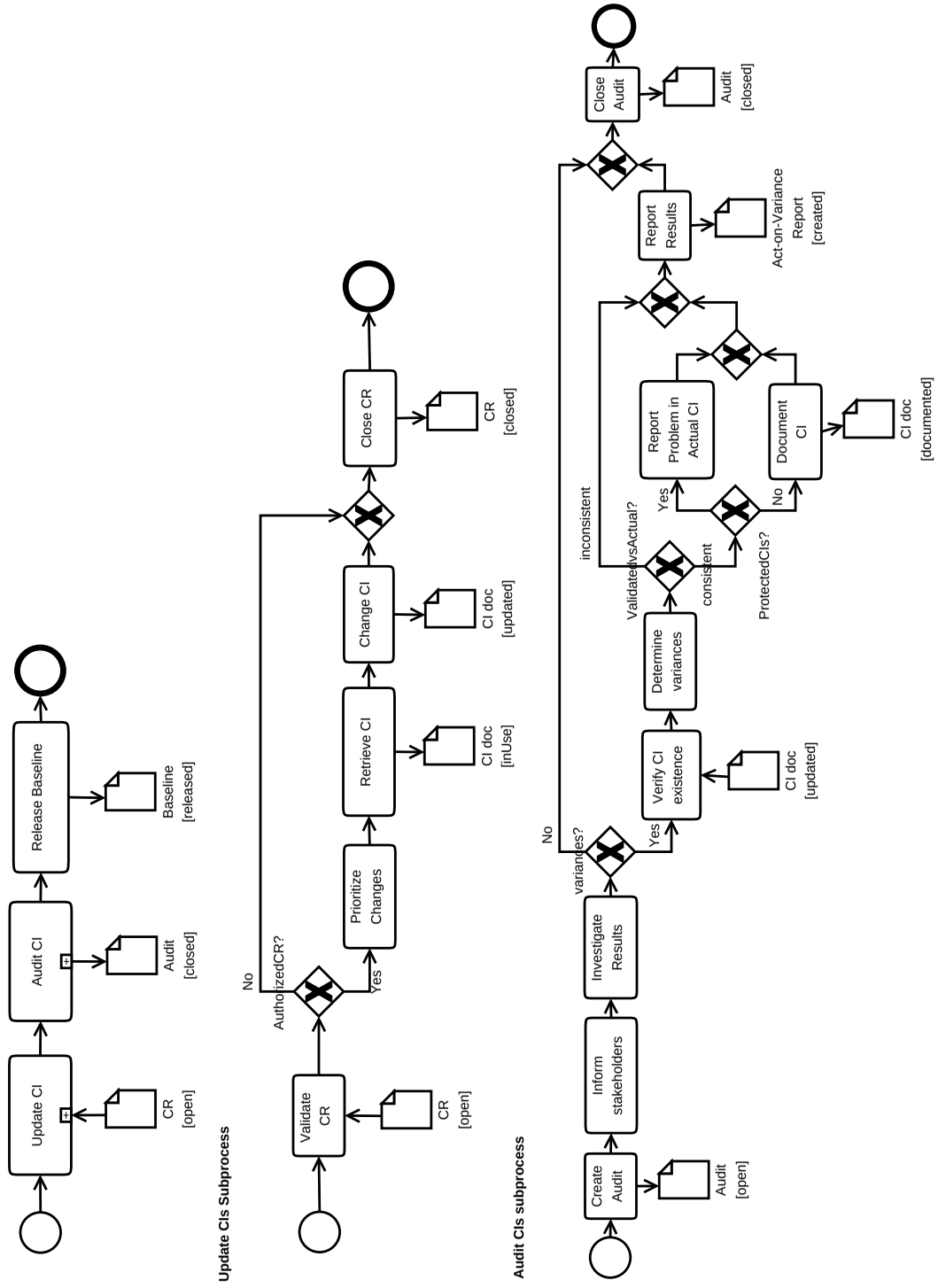


Figure 5.2: EX2 : An interpretation for IBM CCM Process

(DSR) for this example is given in the attached Maude code files (See Appendix A for details). We will refer to this example as **EX2** in the following discussion.

5.2 Property Specifications

In this section, we introduce the Linear Temporal Logic notation, in Section 5.2.1, which we use in defining the properties later. The CMMI Configuration Management process area requirements are formalized in LTL in Section 5.2.3 proposing an application of the BPMN semantics introduced in Chapter 3 and the predicates defined in Section 5.1.1. The approach uses Maude's LTL model checker to perform the compliance checking after that in Section 5.3.

In order to formalize the mapping from the CMMI requirements into LTL properties, we are using the Compliance Request Language (CRL) [32], where the compliance constraints are represented by *compliance patterns* based on the property specification patterns for finite-state verification in [29]. The compliance patterns are classified into four sub-classes of patterns, namely *atomic*, *resource*, *composite*, and *timed*. As long as we are not considering the timed (i.e. into *timed* patterns) and collaboration (i.e. into *resource* patterns) properties of BPs in this stage of formalization, we will focus on the atomic and composite patterns in mapping the CMMI requirements into LTL properties as will be discussed in the Sections 5.2.3. First, we introduce the LTL in Section 5.2.1, and second, the atomic and composite patterns in Section 5.2.2 followed by the mapping rules from atomic patterns to LTL. Third, in Section 5.2.3, the CMMI properties are formalized using the compliance patterns and then mapped into LTL properties.

5.2.1 Linear Temporal Logic (LTL)

Temporal logic is used to specify properties related to infinite behaviour; e.g. a behaviour that depends on certain object state occurrence or disappearance through the traces. It allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens). There are different temporal logics that are used to reason about temporal properties for systems [18, 96]; we focus on linear temporal logic [18, 52], because of its

intuitive appeal [96, 6, 32], widespread use, and well-developed proof methods and decision procedures [20].

Although they are expressively incomparable as they represent two distinct views of time, an interesting comparison between Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) has been introduced in [96]. The study shows that on the verification side, CTL is more difficult than LTL due to the branching nature of CTL. However, the main advantage of CTL over LTL is its computational complexity [96, 32]. In [96], the author argued that this advantage for CTL is valid under worst case scenarios, which are unlikely in real life applications. In [32], the author provided a comparison between different logics used in defining the compliance requirements and argued that using LTL is preferable agreeing with [96] and in order to address the usability concern of LTL, they introduced the Compliance Request Language (CRL) [32] as a high-level pattern-based specification language that enables the abstract specification of legal and organizational compliance requirements. In addition, the approach in [6] was designed to use the Past Linear Temporal Logic (PLTL), i.e. PLTL extends the LTL to allow for representing the past events, as property specification language beside representing the compliance requirements patterns using a graphical query language (Q-BPMN).

As we are focusing on the compliance problem, and due to the domain of compliance requirements as CMMI which is more concerned with process improvement more than the legal and contractual aspect of the business, we are going to use the LTL as the property specification language. Moreover, Maude has its own LTL model checker which accepts Maude modules as system modules and LTL formulae as properties.

An LTL formula consists of: the atomic propositions (state labels $p \in AP$), the Boolean connectors (like conjunction $\wedge$, and negation $\neg$), and two basic temporal modalities ($\bigcirc$ or next) and (U or until) [9].

Definition 5.2.1. (**Abstract Syntax of LTL**) LTL formulae over the set of AP of atomic proposition are formed according to the following grammar: $\varphi ::= \top \mid \bot \mid p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 U \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \Diamond\varphi \mid \Box\varphi \mid \varphi_1 R \varphi_2 \mid \varphi_1 W \varphi_2$ where $p \in AP$

Most of the LTL boolean and temporal connectives can be defined in terms of the minimal set of connectives ($\top$, p, $\neg\varphi$, $\varphi_1 \vee \varphi_2$, $\bigcirc\varphi$, $\varphi_1 U \varphi_2$) as explained above. Some temporal properties are:

- Safety properties: something bad will not happen. For example, $\Box\neg(status(\text{"}CR\text{"}, closed) \wedge status(\text{"}CR\text{"}, open))$ specifying that for a change request CR data object status will not be open and closed at the same time.

- Liveness properties: something good will happen. For example, $\Diamond executed(\text{"}CheckAuthorization\text{"})$, for specifying that Authorization checking will *eventually* be executed.

- Fairness properties: if something is attempted arbitrarily infinitely often, then it must eventually succeed, e.g. $\Box\Diamond status(\text{"}CR\text{"}, open) \rightarrow \Box\Diamond status(\text{"}CR\text{"}, closed)$ to specify that whenever generally eventually a change request is open, it is generally eventually closed.

The goal is to check if the model satisfies each property by solving the satisfiability relation (Equation 2.1) using Maude's LTL model checker. In this work, we use the LTL to formalize the properties of the CMMI requirements in order to check them against the well-formed BPMN model representing the configuration management process area.

5.2.2 Compliance Patterns

Based on [29, 64, 33], *atomic patterns* are used to describe the requirements that involve basic occurrence and ordering of BP elements, while *composite patterns* are used to enable nesting of multiple patterns, i.e. complex requirements, through the Boolean logical operators (Not, And, Or, Xor, Imply, and Iff). For example, *P CoExists Q* pattern is an *Imply* composition of *P Exists* Implies *Q Exists*, which indicates that the presence of *P* mandates that *Q* is also present, given *P*, *Q*, *S*, and *T* as operands representing a BP element, which is then put into a suitable predicate to build the LTL formulae. Table 5.1 summarizes a relevant subset of the compliance patterns developed in [29, 33] and the mapping from these patterns into LTL formulae where *P*, *Q*, *S*, and *T* are operands that indicate BP elements (e.g. activity, data object).

In Table 5.1, the following are occurrence atomic patterns; *P isUniversal* pattern specifies that *P* should always hold throughout the BP model, and *P Exists* pattern specifies that *P* must hold at least once within the BP model. The second set of patterns are the order atomic patterns, and they appear in Table 5.1 as, *P Precedes Q* pattern which

Table 5.1: Compliance Patterns Mapped into LTL [29, 33]

Compliance Pattern	**LTL Formula**
Atomic Patterns: Occurrence Patterns	
P isUniversal [29]	$\Box P$
P Exists [29]	$\Diamond P$
Atomic Patterns: Order patterns	
P Precedes Q [29]	$\neg Q\ W\ P$
P Precedes (S,T) [33]	$(\Diamond(S \wedge O\Diamond T)) \rightarrow (\neg S\ U\ P)$
(S,T) *Precedes P* [33]	$(\Diamond P \rightarrow (\neg P\ U\ (S \wedge \neg P \wedge O(\neg PUT))))$
P LeadsTo Q [29]	$\Box(P \rightarrow \Diamond Q)$
Composite Patterns	
P CoExists Q [33]	$\Diamond P \rightarrow \Diamond Q$

indicates that Q must always be preceded by P, *P Precedes* (S,T) (or $Chain-Precedes$) pattern specifies that a sequence of S,T must be preceded by P, while P, (S,T) *Precedes P* (another form of $Chain-Precedes$) pattern indicates that P must be preceded by a sequence of S,T, and *P LeadsTo Q* pattern indicates that P must always be followed by Q. The third set of patterns are the composite patterns, where two or more of the atomic patterns are used to represent them, such as: *P CoExists Q* pattern requires that presence of P mandates that Q is also present.

An expression built from compliance patterns and operands has a direct mapping to LTL formulas [33]. The formal description of the CRL grammar defining its syntax can be found in [32]. The second column in Table 5.1 gives a brief idea about the mapping from the compliance patterns into LTL formulae which are used to formally present the CMMI-CM requirements into LTL properties. In the next subsection, we introduce a novel mapping from the CMMI-CM requirements into LTL formulae through the compliance patterns introduced in this subsection.

5.2.3 CMMI-CM in LTL

In this subsection, the CMMI Configuration Management (CMMI-CM) process area requirements are interpreted using the compliance patterns introduced in Section 5.2.2, and then mapped into LTL formulae according to the mapping represented in Table 5.1. CMMI-CM has three specific goals (i.e. SG1, SG2, and SG3) and seven specific practices (i.e. SP1.1, SP1.2, SP1.3, SP2.1, SP2.2, SP3.1, and SP3.2). The requirements considered below are the sub-practices under each specific practice and acting towards fulfilling one of the specific goals[2]. The numbers in the parentheses are references to the source in [21], for example, (1.2.3) refers to the first specific goal, second specific practice, and third sub-practice in CM Process Area. In the following, the CMMI-CM requirements are categorized according to their specific practice and addressing related elements in the model.

In Table 5.2, a subset of the sub-practices under SP1.1, SP2.2, SP3.1 and SP3.2 which are related to the CI document statuses in the process are modelled using compliance patterns and then mapped into LTL formulae. The first set of these requirements specifies the *existence* of a list of activities regarding identifying the CI by selecting them based on pre-defined criteria, assigning them IDs, specifying their key feature, when they should enter the Configuration Management System (CMS), the stakeholders and owners involved, and define the relationships among the existing CIs. Therefore, they are represented using *Exists* pattern for each one of them with the conjunction of their existence predicates. The second set of requirements explains the need to document the CIs every time they are used, which is modelled using *LeadsTo* pattern. The third set of requirements specifies that each CI included in the baseline should be recognizable (i.e. documented into the CMS). The *CoExists* pattern is used here to emphasis on the necessity of the existence of both operands, i.e. the documented CI and the activity of releasing baseline. Finally, the fourth set of practices specifies the ability to retrieve and change CIs before they are documented. Therefore, they are represented using the *ChainPrecedes* pattern to identify that the CIs are retrieved, changed and then documented.

[2]More details about the CMMI-CM can be found in Appendix B, where we attach the CM section in the CMMI-DEV 1.3 document [21] for the reader reference for the sake of making this thesis self-contained.

Table 5.2: CMMI-CM CI Requirements mapped using compliance patterns

CMMI-CM Requirements	**Elements Involved**
(1.1.1) Select CIs based on the pre-defined criteria.	***IdentifyCIs***
(1.1.2) Assign unique identifiers to CIs.	***AssignCIids***
(1.1.3) Specify the important characteristics of each CI.	***SpecifyCharacteristics***
(1.1.4) Specify when each CI is placed under CMS.	***SpecifyDate***
(1.1.5) Identify the owner responsible for each CI.	***IdentifyOwners***
(1.1.6) Specify relationships among CI.	***DefineRelationships***
Pattern-based Expression : $IdentifyCIs$ ***Exists*** $\wedge$ $AssignCIids$ ***Exists*** $\wedge$ $SpecifyCharacteristics$ ***Exists*** $\wedge$ $SpecifyDate$ ***Exists*** $\wedge$ $IdentifyOwners$ ***Exists*** $\wedge$ $DefineRelationships$ ***Exists***	
LTL(1) : $\Diamond$executed($IdentifyCIs$) $\wedge$ $\Diamond$executed($AssignCIids$) $\wedge$ $\Diamond$executed($SpecifyCharacteristics$) $\wedge$ $\Diamond$executed($SpecifyDate$) $\wedge$ $\Diamond$executed($IdentifyOwners$) $\wedge$ $\Diamond$executed($DefineRelationships$))	
(2.2.3) Check in and check out CIs in the CMS. (3.1.6) Revise the status of each CI. (3.2.2) Confirm that CM records correctly identify CIs.	***CIdoc:inUse*** ***CIdoc:documented***
Pattern-based Expression : ($CIdoc$,inUse) ***LeadsTo*** ($CIdoc$,documented)	
LTL(2) : $\Box$(status($CIdoc$,inUse) $\rightarrow$ $\Diamond$status($CIdoc$,documented)	
(3.1.1) Record CM actions so each CI's status is known.	***CIdoc:documented*** ***ReleaseBaseline***
Pattern-based Expression : ($CIdoc$,documented) ***CoExists*** $ReleaseBaseline$	
LTL(3) : $\Diamond$status($CIdoc$,documented) $\rightarrow$ $\Diamond$executed($ReleaseBaseline$)	
(1.2.2) Store and retrieve CIs in a CMS. (1.2.5) Store and recover archived versions of CIs. (1.3.3) Document the set of CIs that are in a baseline.	***RetrieveCI*** ***ChangeCI*** ***DocumentCI***
Pattern-based Expression : ($RetrieveCI$,$ChangeCI$) ***ChainPrecedes*** $DocumentCI$	
LTL(4) : $\Diamond$ executed($DocumentCI$) $\rightarrow$ ($\neg$executed($DocumentCI$) U (executed($RetrieveCI$) $\wedge$ $\neg$executed($DocumentCI$) $\wedge$ O($\neg$executed($DocumentCI$) U executed($ChangeCI$))))	

In Table 5.3, a subset of the sub-practices under SP1.2, SP1.3 and SP2.2 which are related to obtaining authorized access before releasing baselines. The requirements are represented using compliance patterns and then mapped into LTL formulae. These practices specify the necessity of having the appropriate authorized access in order to be able to create or release the CIs baselines. Therefore, they are represented using the *Precedes* pattern to identify that the CIs baselines are created or released only if the appropriate authorized access is granted. The second set of requirements specifies that authorization to change CIs is followed by being able to share the information that a CI is being used/updated in the CMS. Hence, the *LeadsTo* pattern in used to model this requirement.

Table 5.3: CMMI-CM Access Requirements mapped using compliance patterns

CMMI-CM Requirements	**Elements Involved**
(1.2.1) Establish a mechanism to manage control levels. (1.2.3) Provide access control to CMS. (1.3.1) Obtain authorization before releasing baselines. (2.2.2) Obtain authorization to enter changed CIs into CMS.	***Authorized?,YES*** ***ReleaseBaseline***
Pattern-based Expression : (*Authorized*?:YES) ***Precedes*** *ReleaseBaseline*	
LTL(5) : $\neg$ executed(*ReleaseBaseline*) *W* conditionGuard(*Authorized*? : "YES")	
(1.2.4) Share and transfer CIs in the CMS.	***Authorized?,YES*** ***CIdoc:inUse***
Pattern-based Expression : (*Authorized*?:YES) ***LeadsTo*** (*CIdoc*,inUse)	
LTL(6) : $\Box$(conditionGuard(*Authorized*?, "YES") $\rightarrow \Diamond$status(*CIdoc*,inUse))	

Table 5.4 presents two subsets of the sub-practices under SP1.3, and SP3.1 which are related to baselines. The first set of requirements is considering that documenting the CIs into the CMS must exist and releasing the related baseline exists. The second set of requirements in the table refer to the fact that the released baseline should be made readily available for stakeholders. The requirements are represented using compliance patterns and then mapped into LTL formulae. Therefore, the first set are represented using the *CoExists* pattern to identify that the documenting CIs mandates the releasing of the updated baseline. For the second set of requirements, the *LeadsTo* pattern is used

to specify that releasing a baseline should be followed by making it accessible for the relevant stakeholders.

Table 5.4: CMMI-CM Baseline Requirements mapped using compliance patterns

CMMI-CM Requirements	**Elements Involved**
(1.3.2) Release baselines only from CIs in the CMS. (3.1.3) Identify the version of CIs in a particular baseline.	***ReleaseBaseline*** ***DocumentCIs***
Pattern-based Expression : (*DocumentCIs*) ***CoExists*** *ReleaseBaseline*	
LTL(7) : $\Diamond$executed(*DocumentCI*) $\rightarrow \Diamond$executed(*ReleaseBaseline*)	
(1.3.4) Make the current set of baselines readily available. (3.1.2) Ensure that relevant stakeholders have access to CIs. (3.1.5) Specify the latest version of baselines. (3.1.4) Describe differences between successive baselines.	***MakeBaseline*** ***Available*** ***Baseline,released***
Pattern-based Expression:(*Baseline*,released) ***LeadsTo*** *MakeBaselineAvailableToRead*	
LTL(8):$\Box$ (status(*Baseline*,released) $\rightarrow \Diamond$ executed(*MakeBaselineAvailableToRead*))	

In Table 5.5, four subsets of the sub-practices under SP2.1, SP2.2, and SP3.2 which are related to the change requests (CR) are mapped. The first set specifies that each opened CR should be closed later in the BPs. This is modelled using the pattern *LeadsTo* and mapped into the corresponding LTL formula. The second set of requirements in the table refer to that releasing baseline must always be preceded by validating the CR as an authorized CR. It is modelled using *Precedes* pattern. The third set of requirements represents the necessity of prioritizing the available CRs, and hence, modelled using *Exists* pattern and mapped into the LTL formula shown in the table. The final requirement specifies that opening a CR leads to investigating the variances of the updated CIs. Therefore, the *LeadsTo* pattern is used.

In Table 5.6, a subset of the sub-practices under SP1.2, SP2.2 and SP3.2 which are related to the audit document is modelled. They are specifying that an audit document should be tracked from opening until closure in the CMS. The audit normally contains information documenting, evaluating, and confirming the CI status in the CMS. The audit requirements are represented using *LeadsTo* pattern and then mapped into LTL formulae to identify that the an audit document should be closed after it is opened.

Table 5.5: CMMI-CM CR Requirements mapped using compliance patterns

CMMI-CM Requirements	**Elements Involved**
(2.1.1) Initiate and record CRs in the CMS. (2.1.5) Track the status of CRs to closure. (2.2.1) Control changes to CRs in its lifecycle.	***CR,open*** ***CR,closed***
Pattern-based Expression : (*CR*,open) ***LeadsTo*** (*CR*,closed)	
LTL(9) : $\Box$(status(*CR*,open) $\rightarrow \Diamond$status(*CR*,closed))	
(2.1.2) Analyze the impact of CRs. (2.1.4) Review CRs with relevant stakeholders.	***AuthorizedCR?:YES*** ***ReleaseBaseline***
Pattern-based Expression : (*AuthorizedCR*?:YES) ***Precedes*** *ReleaseBaseline*	
LTL(10):$\neg$executed(*ReleaseBaseline*) W conditionGuard(*authorizedCR*?:YES)	
(2.1.3) Categorize and prioritize CRs.	***PrioritizeChanges***
Pattern-based Expression : *PrioritizeChanges* ***Exists***	
LTL(11) : $\Diamond$executed(*PrioritizeChanges*)	
(3.2.4) Confirm the correctness of approved CRs.	***CR:open*** ***InvestigateResults***
Pattern-based Expression : (*CR*,open) ***LeadsTo*** *InvestigateResults*	
LTL(12):$\Box$(status(*CR*,open) $\rightarrow \Diamond$executed(*InvestigateResults*))	

Table 5.6: CMMI-CM Audit Requirements mapped using compliance patterns

CMMI-CM Requirements	**Elements Involved**
(1.2.7) Create CM reports from the CMS. (2.2.5) Record changes to CIs and reasons for them. (3.2.6) Track action items from the audit to closure.	***Audit:open*** ***Audit:closed***
Pattern-based Expression : (*Audit*,open) ***LeadsTo*** (*Audit*,closed)	
LTL(13) : $\Box$(status(*Audit*,open) $\rightarrow \Diamond$status(*Audit*,closed))	

In Table 5.7, a subset of the sub-practices under SP1.2, SP2.2 and SP3.2 which is related to the variance handling is modelled. The first set of requirements specifies the necessity of working on the variances once they are occur in order to preserve the structure of the CMS components. Therefore, the *LeadsTo* compliance pattern is used to represent that determining the variances should follow deciding the existence of them. The second set of requirements specifies that once a change request is open, then the variances are investigated for the possible solutions. The pattern *Precedes* is used to model the requirement and the corresponding LTL formula is produced.

Table 5.7: Variance Handling Requirements mapped using compliance patterns

CMMI-CM Requirements	**Elements Involved**
(1.2.8) Preserve the contents of CMS. (1.2.9) Revise the CM structure as necessary. (3.2.5) Confirm compliance with standards.	***variances?:Yes*** ***DetermineVariances***
Pattern-based Expression : (*variances*?:Yes) ***LeadsTo*** *DetermineVariances*	
LTL(14):□(conditionGuard(*variances*:Yes) → ◇executed(*DetermineVariances*))	
(2.2.4) Ensure changes have not compromised CMS. (3.2.1) Assess integrity of Baseline. (3.2.3) Review the structure and integrity of CMS.	***CR,open*** ***InvestigateResults***
Pattern-based Expression : (*CR*,open) ***Precedes*** *InvestigateResults*	
LTL(15) : ¬ executed(*InvestigateResults*) *W* status(*CR*,open)	

The property specifications (φ) for the CMMI-CM requirements are now mapped into LTL. In the following section, they are checked against the example models (EX1 in Figure 5.1, and EX2 in Figure 5.2) and the results are discussed. Notice the numbers beside LTL in the Tables 5.2, 5.3, 5.4, 5.5, 5.6, 5.7 is referring to the property number which will be used to link the results with the corresponding LTL property. Therefore, the plan is to check if the model satisfies each property in (φ) by solving the satisfiability relation (Equation 2.1) using Maude's LTL model checker. A list of the above LTL properties is given in Table 5.8.

Table 5.8: CMMI-CM Requirements into LTL

CMMI-CM Requirements
Configuration Items Requirements
LTL(1) : $\Diamond$executed(*IdentifyCI*) $\wedge$ $\Diamond$executed(*AssignCIid*) $\wedge$ $\Diamond$executed(*SpecifyCharacteristics*) $\wedge$ $\Diamond$executed(*SpecifyDate*) $\wedge$ $\Diamond$executed(*IdentifyOwners*) $\wedge$ $\Diamond$executed(*DefineRelationships*))
LTL(2) : $\Box$(status(*CIdoc*, *inUse*) $\rightarrow$ $\Diamond$status(*CIdoc*, *documented*)
LTL(3) : $\Diamond$status(*CIdoc*, *documented*) $\rightarrow$ $\Diamond$executed(*ReleaseBaseline*)
LTL(4) : $\Diamond$ executed(*DocumentCI*) $\rightarrow$ ($\neg$executed(*DocumentCI*) *U* (executed(*RetrieveCI*) $\wedge$ $\neg$executed(*DocumentCI*) $\wedge$ *O*($\neg$executed(*DocumentCI*) *U* executed(*ChangeCI*))))
Access Control Requirements
LTL(5) : $\neg$ executed(*ReleaseBaseline*) *W* conditionGuard(*Authorized*?: *"YES"*)
LTL(6) : $\Box$(conditionGuard(*Authorized*?: *"YES"*) $\rightarrow$ $\Diamond$status(*CIdoc*, *inUse*))
Baselines Requirements
LTL(7) : $\Diamond$executed(*DocumentCI*) $\rightarrow$ $\Diamond$executed(*ReleaseBaseline*)
LTL(8) : $\Box$(status(*Baseline*, *released*)$\rightarrow$ $\Diamond$ executed(*MakeBaselineAvailableToRead*))
Change Requests Requirements
LTL(9) : $\Box$(status(*CR*, *open*) $\rightarrow$ $\Diamond$status(*CR*, *closed*))
LTL(10) : $\neg$executed(*ReleaseBaseline*) *W* conditionGuard(*authorizedCR*?: *"YES"*)
LTL(11) : $\Diamond$executed(*PriotarizeChanges*)
LTL(12) : $\Box$(status(*CR*, *open*) $\rightarrow$ $\Diamond$executed(*InvestigateResults*))
Audit Requirements
LTL(13) : $\Box$(status(*Audit*, *open*) $\rightarrow$ $\Diamond$status(*Audit*, *closed*))
Variance Handling Requirements
LTL(14) : $\Box$(conditionGuard(*variances* : *"Yes"*) $\rightarrow$ $\Diamond$executed(*DetermineVariances*))
LTL(15) : $\neg$ executed(*InvestigateResults*) *W* status(*CR*, *open*)

5.3 Model Checking Procedure

For the CM process area, satisfying all the included specific goals supports, along with other CMMI process areas (which are not discussed in this work), granting maturity level 2 (ML2) to the process. Hence the requirements for configuration management should be satisfied by the BP model according to the proposed requirement grading scheme. In this section, we discuss the results of checking the properties formalized in Section 5.2.3 against the two introduced example models (i.e. EX1 and EX2). A compliance grading scheme is introduced to explain the model checking results according to the domain of application (i.e. CMMI-CM compliance checking). Based on considering compliance checking a model checking problem (c.f. Section 5.1), the relation $M \models \varphi$ simulates the formal checking process. In the following, we introduce the compliance grading scheme to interpret the results in Subsection 5.3.1. After that we discuss some features of the properties which might affect the model checking results in Subsection 5.3.2.

5.3.1 Compliance Grading Scheme

For the purpose of identifying the compliance results, we introduce the following requirements satisfaction grading scheme. It is summarized in Table 5.9, where each LTL property has been assigned a weight based on the number of sub-practices it represents based on the mapping presented in Tables 5.2, 5.4, 5.3, 5.6, 5.5, 5.7. If the summation of the total weights of the properties which the model scores less than 12 points, then the model is *Not Compliant*, if between 12 and 24, then the model is *Partially Compliant*, if between 25 and 35, then the model is *Largely Compliant*, and if at least 36 point of weights are gained, then the model is *Fully Compliant*. The total number of

Table 5.9: Requirements Satisfaction Grading Scheme

Grade Label	**Grade**%	**Grade Points Explanation**
Fully Compliant	90%-100%	at least 36 weight points
Largely Compliant	60%-less than 90%	25-35 weight points
Partially Compliant	30%-less than 60%	12-24 weight points
Not Compliant	0%-less than 30%	less than 12 weight points

sub-practices in the CMMI-CM process area is 41 sub-practice. Hence, we can conclude that if a model M satisfies the property LTL(2), then it will score 3 weight points. The total number of weight points a model scores determines which compliance grade it should be assigned based on Table 5.9.

5.3.2 Spurious Properties

The result of model checking is either *true* or a *counterexample*. In the first case (i.e. *true*), the property φ is satisfied by the model M. The other case (i.e. *counterexample*) specifies that the property φ is *not* satisfied by at least one trace of the model M execution. In this work, the first case can be interpreted as a positive property compliance indicator while the other case can be interpreted as a negative property compliance indicator. Therefore, $M \models \varphi$ results in *true* can be explained as : the model M is compliant with the requirement formally represented as φ. Similarly, $M \not\models \varphi$ results in a *counterexample* can be explained as : the model M is NOT compliant with the requirement formally represented as φ. However, having counterexamples in our case can be misleading. It might not be a result of the model does not satisfy the property, but the model lacking the objects which the property define their temporal relationship.

Irrelevant (*spurious*) properties normally result from the absence of one or more objects that are involved into the property specified, or due to the use of different naming words specific to the company. The process modeller has to decide if the elements represented by these missing objects are important for the process and are unintentionally missed, then they have to be added to the process and restart the property checking. In our approach, we limit the application of model checking to properties which are not spurious with respect to the documents or activities it checks. On one hand, this help to avoid spurious results. On the other hand, it reduces the computation cost incurred when a model checking procedure is performed for a spurious property. Therefore, we manipulated our model checking procedure to consider this situation and defined a third possible model checking result as (`DocumentDoesNotExist`).

The main checking operator used here is `checkProp` which takes a formula and a BP model and performs the model checking using operator `modelCheck` if and only if the objects being tested in the formula are into the BP model.

```
op DocumentDoesNotExist : -> ModelCheckResult .
op checkProp : Formula TraceObjectSet -> ModelCheckResult .
ceq checkProp (F, A) = modelCheck(A, F) if CE(F, A) == true .
eq checkProp (F, A) = DocumentDoesNotExist [owise] .
```

The operator `CE` checks if all the objects involved in a property do already exist in the model being checked or not as defined below. The operator takes a property (`Formula`) representing one of the fifteen LTL formulae of the CMMI-CM in Table 5.8 and the BP model (`TraceObjectSet`) represented by variable `A`. It returns *true* if the objects exist in the BP model and *false* if at least one of the objects does not exist. As an example part of the definition, we provide the equation for the fifth LTL property in Table 5.8 which includes activity `Release Baseline` and guard variable `Authorized?`.

```
op CE : Formula TraceObjectSet -> Bool .
ceq CE(F, A) = true
if (F == LTL(5)) /\ ("Release Baseline" into A) /\ (Authorized? into A) .
...
eq CE(F,A) = false [owise] .
```

The above definition uses function `into` which is defined to take a string value for an object name (i.e. `String`) or a defined variable name as part of guard expression (i.e. `Variable`) and a process model. It returns *true* if there is an object with the same name (or a guard uses the input variable name as part of its expression) into the process model. Otherwise, it returns $false$, i.e. if the object (or a guard expression variable name) is not in the process model.

```
op _into_ : String TraceObjectSet -> Bool .
op _into_ : Variable TraceObjectSet -> Bool .
eq S1 into (CVcol * << < X : K | name : S1 ; AS1 > , B >>) = true .
eq S1 into A = false [owise] .
eq V1 into (((V1 : S1) .. CVcol) * B) = true .
eq V1 into A = false [owise] .
```

Another issue here is that properties like LTL(6), LTL(8), LTL(12), LTL(13) and LTL(14) are presented using (*P LeadsTo Q*) pattern, which is mapped into LTL as ($\Box(P \rightarrow \Diamond Q)$) and its result depends on the *implies* operator (i.e. $\rightarrow$) result. According to the

($\rightarrow$) truth table, the result is *true* if the two operands are *true*, if the two operands are *false*, and if the first is *false* and the second is *true*. That is, if the two elements being checked are not included into the model, and hence their predicates result in *false*[3], the model checking result of the properties would still *true*. For example, in property LTL(12), the predicate *executed*(”*InvestigateResults*”) would be rewritten to *false* following the `[owise]` statement defined in Section 5.1.1 while the object is not in the model.

5.3.3 Results Representation

As the example models EX1 and EX2 specify releasing baselines based on controlling change requests and not adding new CIs, we expect some of the 15 properties modelled for the CMMI-CM to be spurious. However, in order to make the approach complete, i.e. in terms of checking all the requirements in the CMMI-CM process area, we are checking all the modelled properties here. The main command is (`red checkProp(LTL(5), initial) .`) for the property LTL(5). In Table 5.10, the weight column refers to the number of sub-practices that are represented into the LTL property. Hence, if a model *M* satisfies the property LTL(5), then it will score 4 weight points. According to the proposed grading scheme in Table 5.9 and the model checking results represented in Table 5.10, we can conclude that model EX1 is *Partially Compliant* and model EX2 is *Partially Compliant* as well. Although it seems that the model EX1 is smaller than the model in EX2, but EX1 scored seven more points than the model EX2 in the model checking procedure.

Our approach has been implemented using Maude’s modules and its LTL model checker via the Eclipse platform. Both of the software is free and available to download from the Internet. The cost in terms of time of execution for checking the compliance of EX1 and EX2 is detailed in Table 5.11. The longest rewrite time for a trace in a process is consumed in rewriting from start state to the end state. From the table, we can see that the rewrite time for the properties that we considered spurious is relatively smaller

[3]Note that the predicates definition in Section 5.1.1 has a false case definition to rewrite the predicate into *false* in all other cases than the defined ones. That why operator statement attribute `[owise]` was used.

Table 5.10: Model Checking Results for EX1 and EX2

Property	**Weight**	**EX1**	**Points**	**EX2**	**Points**
LTL(1)	6	`DocumentDoesNotExist`	0	`DocumentDoesNotExist`	0
LTL(2)	3	true	3	true	3
LTL(3)	1	true	1	true	1
LTL(4)	3	true	3	true	3
LTL(5)	4	true	4	`DocumentDoesNotExist`	0
LTL(6)	1	CE1.6	0	`DocumentDoesNotExist`	0
LTL(7)	2	true	2	CE2.7	0
LTL(8)	4	true	4	`DocumentDoesNotExist`	0
LTL(9)	3	true	3	true	3
LTL(10)	2	true	2	CE2.10	0
LTL(11)	1	`DocumentDoesNotExist`	0	true	1
LTL(12)	1	`DocumentDoesNotExist`	0	true	1
LTL(13)	3	`DocumentDoesNotExist`	0	true	3
LTL(14)	4	`DocumentDoesNotExist`	0	CE2.14	0
LTL(15)	3	`DocumentDoesNotExist`	0	CE2.15	0
Total	41	**22**		**15**	
Results		***Partially Compliant***		***Partially Compliant***	

than the rewrite time for the properties passed to the model checker. These properties are LTL(1,11,12,13,14,15) for EX1 and LTL(1,5,6,8) for EX2. The total number of rewrites for checking the 15 properties for EX1 was 105906 rewrites in total of about 673 ms and for EX2 was 12339 in total of about 380 ms. Given the fact that configuration management processes are normally of the same size as our two examples EX1 and EX2, one can see that these results as an indication of the applicability of the designed tool to a wide range of CM processes.

Following the approach introduced in Chapter 1, at this stage, the modeller has an idea about what requirement exactly the process violates. Moreover, the modeller can decide if certain properties are relevant to the process being checked or can be ignored. For example, a possible modification in model EX2 could be to check the authorization for

Table 5.11: Summary Rewrite time for EX1 and EX2

	EX1 (rewrites in ms)	**EX2** (rewrites in ms)
Process Exec Time	204 in 1ms	95 in 5ms
Property	**EX1**(rewrites in ms)	**EX2**(rewrites in ms)
LTL(1)	106 in 0ms	106 in 0ms
LTL(2)	34195 in 230ms	909 in 33ms
LTL(3)	15368 in 73ms	959 in 31ms
LTL(4)	3710 in 37ms	134 in 5ms
LTL(5)	91 in 1ms	103 in 0ms
LTL(6)	2212 in 22ms	103 in 0ms
LTL(7)	15086 in 75ms	1635 in 44ms
LTL(8)	103 in 0ms	103 in 0ms
LTL(9)	34234 in 228ms	953 in 39ms
LTL(10)	288 in 7ms	1965 in 53ms
LTL(11)	101 in 0ms	117 in 3ms
LTL(12)	104 in 0ms	971 in 38ms
LTL(13)	103 in 0ms	975 in 33ms
LTL(14)	103 in 0ms	1641 in 51ms
LTL(15)	102 in 0ms	1665 in 50ms
Total$_{ComplianceCheckingTime}$	105906 in 673ms	12339 in 380ms

conducting a change in the configuration management database before it is actually open, used and documented. An authorization check can be added once the process started as in the updated version of EX2 in Figure 5.3. In the model EX2m, the activity "`Make Baseline Available to Read`" has been moved to after the baseline is released, an authorization check is added using the XOR gateway after the process starts and before reading and updating any CIs. After implementing these changes into the model EX2, the model checking results changes for properties (5,6,8) from `DocumentDoesNotExist` to `true` with added weights of (4,1,4) respectively. Now the model EX2m scores 24 out of 41 and according to the grading scheme in Table 5.9 the model is still partially compliant with CMMI-CM requirements, however, with a higher score points. This is an

example of how some changes in the designed model checked with our approach can easily change its compliance grade score points to a higher one giving better chances in an appraisal.

Table 5.12: Summary compliance checking results for EX2m

		EX2m (rewrites in ms)
	Exec Time	106 in 3ms
Property	**MC Result**(points)	**EX2m** (rewrites in ms)
LTL(1)	`DocumentDoesNotExist`(0)	106 in 0ms
LTL(2)	true (3)	1124 in 47ms
LTL(3)	true (1)	945 in 31ms
LTL(4)	true (3)	144 in 6ms
LTL(5)	true (4)	2123 in 60ms
LTL(6)	true (1)	1147 in 45ms
LTL(7)	CE2m.7 (0)	1752 in 52ms
LTL(8)	true (4)	919 in 31ms
LTL(9)	true (3)	928 in 34ms
LTL(10)	CE2m.10 (0)	1455 in 27ms
LTL(11)	true (1)	126 in 3ms
LTL(12)	true (1)	1186 in 47ms
LTL(13)	true (3)	1190 in 40ms
LTL(14)	CE2m.14 (0)	1758 in 56ms
LTL(15)	CE2m.15 (0)	1782 in 55ms
	Points (24)	**Total**$_{ComplianceCheckingTime}$: 16685 in 534ms

An interesting challenge is the naming of the objects which may be different from the pre-defined properties. It can be due to the use of some company's jargon to name the activities and/or properties, hence, there might be a *misinterpretation* in the model checker results side about same activities with different names in the model and the properties. In the compliance checking research, this problem force most researchers to use manual mapping between activities having the same functionality but differs in names (e.g.[38]). For example, task `Inform Stakeholders` in EX2 which has the same

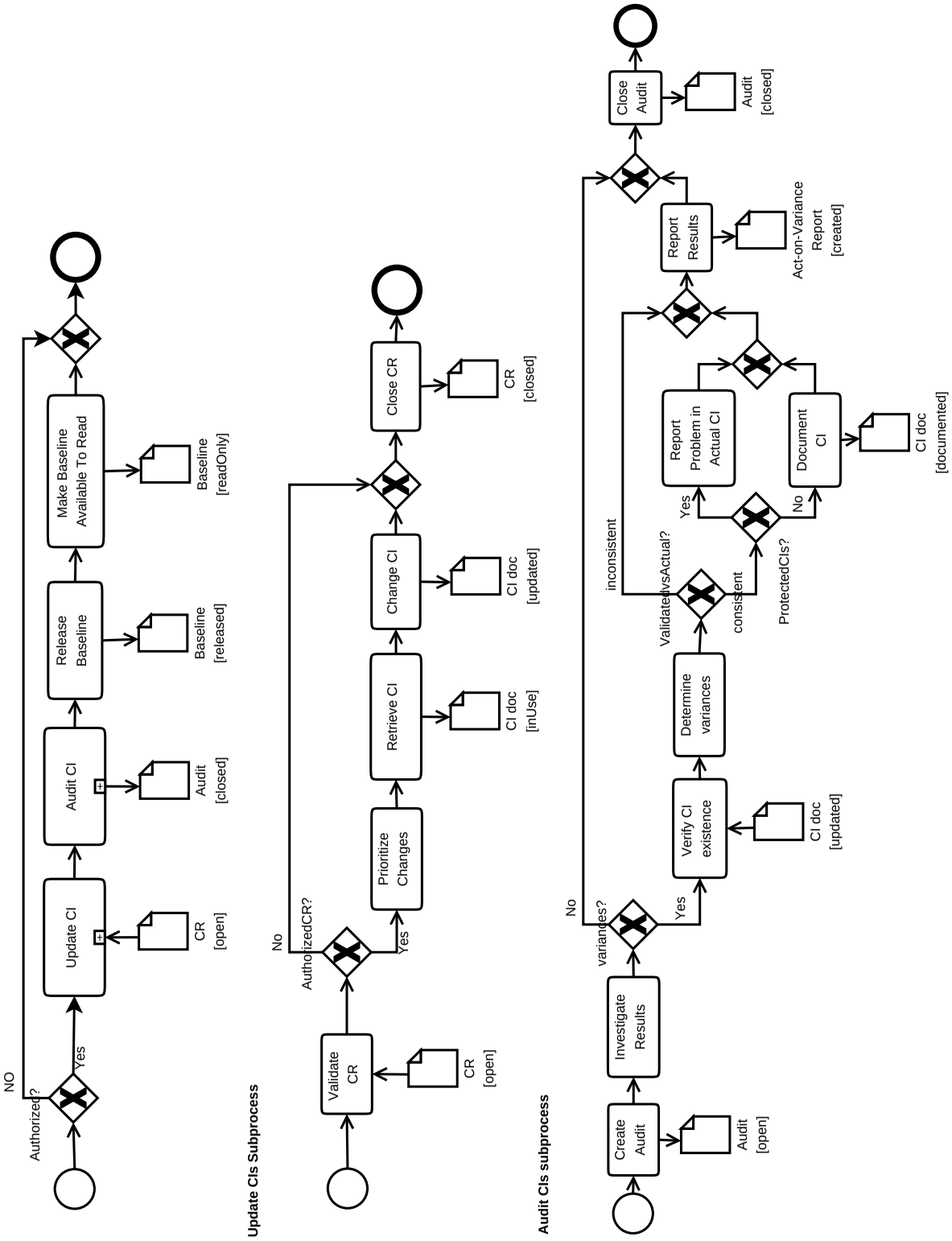


Figure 5.3: EX2m : Model EX2 after update

functionality as `Make Baseline Available to Read` but obviously different names. Therefore, in the updated EX2 process model in Figure 5.3, the activity name has been changed to match the property. Alternatively, one can keep the process model naming convention and update the LTL properties accordingly.

The CMMI appraisal is concerned with two aspects of the BP. One is the evidences of the existence of some documentations and reports that record every step in the process and form part of its inputs and outputs as well in order to produce a product or a service. The other aspect is confirming that the BP is following a well-planned procedures which guarantee its sustainability, accessibility, safety and development. As we are proposing a pre-appraisal approach for CMMI-CM compliance checking, we focused on these two aspects and proposed how to formally assess them in the designed BP, which in turn will allow for a confident application for a SCAMPI A appraisal. The first aspect, i.e. documents checking, can be automatically conducted for the designed BP in the form of checking if certain set of documents are included into the BP. This ha been considered in the model checking procedure by limiting the application of the properties checking to the properties which objects are in the model being checked. Therefore, checking if the documents (as being objects) are represented in the model provides the evidence of their existence in the designed BP. The second aspect, i.e. the process is following a well-planned procedure, is tackled in this chapter by formalizing the CMMI-CM requirements into LTL formulae through compliance patterns to formally check them using Maude LTL model checker.

5.4 Chapter Summary

In this chapter we introduced a novel formalization for the CMMI-CM process area requirements in LTL based on the compliance patterns [29, 33]. This is followed by using Maude LTL Model Checker to check if the introduced two example models of CM processes are compliant with the formalized properties to conclude our approach with analysing and reasoning about the model checking results. Next chapter will present the related work.

Chapter 6

Related Work

In this chapter, we introduce the related published work and discuss their relations with the contributions of this thesis. As there have been published research in different areas related to this work, we categorized them in three sections. In Section 6.1, the BPs different formalization and verification approaches are discussed. Then the Maude applications for concurrent processes in Section 6.2. Finally, the related compliance checking approaches are discussed in Section 6.3.

6.1 BP Formalizations and Verification

There are many attempts to transform the BPMN as a graphical language into more formal languages, such as: Petri nets [27], YAWL [107], CSP [104, 105], π-Calculus [74, 75], and graphs [28, 40]. Table 6.1 gives a brief summary of the formalizations compared to our proposed formalization.

BPMN core flow elements have been mapped into Petri nets in [27]. A number of modelling deficiencies have been identified but not solved; such as models with multiple start events, and OR-join gateway semantics. Although they have defined a well-formed BPMN process to facilitate the mapping, classical Petri nets has limitations in representing certain constructs and behaviour [28] (i.e. OR join, deterministic choice). Generally, the resulting models of Petri net mappings are verified using ProM for the absence of dead transitions, deadlocks, and livelocks. Then there is [107] presenting a formal mapping from BPMN to YAWL [91] based on a well-formed subset of BPMN to establish the mapped models into YAWL-nets. However, the OR join gateways and data objects have not been discussed.

Process Algebra Communicating Sequential Processes (CSP) has been used to formalize the syntax and semantics for a subset of the BPMN core flow elements and models into CSP in [104, 105]. Basically, tasks are mapped into CSP processes and flow transitions are mapped into CSP events. A similar notion to well-formed BPMN models is used, which is well-configured sets of well-formed states (WCF), was used to describe a more restricted and well-formed elements. In [104], CSP is used to define the process state-based specification syntax and behavioural semantics for a subset of the BPMN. This facilitates the use of the CSP-based model checker (i.e. FDR) for property analysis of business processes for refinement, soundness and property checking of BPMN models. Nevertheless, the OR join gateways and guard evaluation were not considered as part of the formalization. The work was extended to handle the timed models in [105].

In [74, 75], the workflow patterns [103] have been formalized into π-calculus (i.e. an algebra for modelling concurrent communicating processes) in order to check the soundness of resulting workflow models. In [74], they showed that the π-calculus is indeed able to handle all of the behavioural workflow requirements given by workflow patterns. In [75], process representation in π-Calculus is used to formalize the BPMN

models aiming at verifying the semantics (i.e. using lazy soundness). Nevertheless, the approach did not consider the well-formed property and did not consider data objects and guard evaluation.

In [28, 40], a subset of the BPMN control flow semantics has been formalized as graph rewrite rules. In [28], GrGen, a graph rewrite tool, was introduced for the execution semantics. However, the notion of a well-formed BPMN models has not been used, which allow for models which are not sound. An in-place token-based approach was introduced in [40] for defining BPMN execution semantics in terms of graph transformation rules. The produced semantics is then verified using unit and integration testing. Nevertheless, the data objects and well-formed notion were not used. While the formalization goal was to have a deadlock-free and sound models in [27, 107], the conformance to specifications and visualization were the targets in [28, 40].

An AI approach was followed in [48] to give a logical model for a subset of BPMN elements discussing some correctness criteria, e.g. deadlock freedom, termination and determinism. The approach utilized the notion of well-formed BPMN elements to represent BP models. However, no formalization of the data objects or guard evaluation mechanism were introduced. A formal syntax for the business process work-flow patterns was introduced using Maude in [43]. BPMN was used as an example application domain for using Maude strategies for a catalogue information system. While the approach used Maude strategies, the notion of well-formed process has not been used and the data objects have not been modelled.

In [103], the notion of structured BPMN work-flow was used to address the need to have an OR merge gateway to every OR split to synchronize the flow. This may be similar to our block structure condition for the well-formed BPMN models, however, in our proposed formalization, we require all the well-formed gateways to have the block structure and not only the OR gateways, discussed in Section 3.3 and Section 4.1. Moreover, the idea of well-formed models was used in the Petri nets in [76] where the well-formed elementary system net was defined as a weakly live (i.e. for each transition there exist a reachable marking that enables it), terminable net (i.e from each reachable marking, the final marking can be reached) and having a unique final marking.

In Table 6.1, a summarized comparison between the state-of-the-art research and our

Table 6.1: Comparison of Related Work and Our Contributions

Ref.	[68]	[27]	[107]	[75]	[104]	[28, 40]	[48]	[43]	this work
Formalism	English	Petri Nets	YAWL	π -Calculus	CSP	Graph Trans.	Prolog	Maude	Maude
OR-Join	●	○	○	○	●	●	○	●	●
Guard Evaluation	○	○	○	○	○	○	○	○	●
Data Objects	●	○	○	○	○	○	○	○	●
Well-Structured	●	○	○	○	○	○	○	○	●
Well-Formed	○	●	●	○	○	○	●	○	●
Verification	○	cs	cs	ls	r,cs	c,t	cr	pv	cc

Legend: ○ - (NO), ● - (YES), cs - (classical soundness), ls - (lazy soundness), r - (refinement), c - (conformance to semantic specifications), t - (unit and integration testing), vis - (visualisation), cr - (correctness), pv - (property verification) and cc - (compliance checking).

proposed approach is presented. The criteria in the first column are, in order, the formalization language, modelling OR-join behaviour, guard evaluation, data objects modelling, the notion of well-structured BPMN processes, the notion of well-formed BPMN processes, and the verification/application methods. The formalization language used in these approaches are, in order, natural language English [68], Petri nets [27], YAWL [107], π-calculus [75], CSP [104], Graph transformation [28, 40], Maude [43], BPMN-Q [6], and ours uses Maude. The legend for the table information is: $\circ$: (NO), $\bullet$: (YES), cs: (classical soundness), ls: (lazy soundness), r: (refinement), c: (conformance to semantic specifications), t: (unit and integration testing), vis: (visualisation), pv: (property verification) and cc: (compliance checking). To the best of our knowledge, no approach has introduced a mechanism to evaluate the guards in the gateways as our approach proposes. Moreover, our approach provides a sound semantics for well-formed BPMN models, which may include OR gateways, as described in Chapter 4 beside checking the well-formedness property as part of the semantics as described in Chapter 3. The semantics introduced in this thesis differs from the other related semantics with respect to the formalization language that is used and the prospective use of the semantics in the application domain. We use the Maude as the formalization language and aiming at formalizing the BPMN models into sound processes to check the compliance with process improvement models.

On verifying the formalized business process models, soundness property is proved to be the most checked property among the scanned literature (e.g. [81, 100, 95, 92, 25, 75]). In [81], two structural conflicts in process models were discussed; i.e. deadlock and lack of synchronization. These two important structural properties disappearance from a model indicate that this model is *sound* with respect to the specifications; i.e. soundness = no deadlock + no lack of synchronization [100]. In [95], different types of soundness were discussed for workflow models. *Classical soundness* [92, 102] states that each activity should be on a path from the initial to the final activity, that after the final activity has been reached no other activities should become active, and that there are no unreachable activities. The other notions of workflow soundness ranges from a more relaxed or weaker version (*weak soundness* [53], *lazy soundness* [75], *relaxed soundness* [25]) of the classical soundness to more stronger version (*generalized soundness* [44]).

A formal proof of soundness is in Chapter 4 based on the classical soundness definition. *k-soundness* [44] restricts the workflow net to have a start node and an end node and that each object reachable from start node is on a path to the end node, *Weak soundness* [53] allows unreachable activities. *Relaxed soundness* [25] softens the original soundness notion claiming to be more easily applicable to application-oriented modelling. It states that each activity should be able to participate in the business process, i.e. for each transition there exists a sequence that takes the initial state to the final state without leaving any spare tokens in the net. Hence, it does not avoid situations with dangling tokens or livelocks/deadlocks [95]. *Lazy soundness* [75] requires the end event to be semantically reachable from every node semantically reachable from the start event until the end event has been executed and that the end event is executed exactly once.

6.2 Maude Applications

Maude, as a logical semantics framework is used to define syntax and semantics of a wide range of languages and logics [57], e.g. CCS [98], π-calculus [88], UML [61, 12], Petri nets [87] and BPMN [43].

In [98], an implementation of the CCS operational semantics in Maude is given, where transitions are rewrites and inference rules are conditional rewrite rules, possibly with rewrite rules in the conditions. Moreover, the authors proposed an implementation of the Hennessy-Milner modal logic for describing processes, with comments on extensions to the LOTOS language [99]. In [88], an executable specification of the operational semantics of an asynchronous version of the π-calculus was introduced in Maude using conditional rewrite rules with rewrites in the conditions too.

UML diagrams have been modelled in Maude and formally verified using Maude LTL model checker in [61]. Moreover, on checking the consistency of UML model design, the authors in [12] formally classified inconsistencies, resolution rules conditions and conclusions for UML models using equational logic and applied in Maude. Petri nets were given conceptual and executable rewriting semantics in [87] where they could also be formally analysed and model checked by means of rewriting strategies that explore and analyse at the meta-level the different rewriting computations of a given rewrite

specification.

For domain-specific languages, Maude proved to be efficient in providing formal rewriting semantics for them (e.g. [14, 78]). The fact that Maude provides object oriented facilities that can be used to implement metamodels and models has been experimented by several research [78, 98, 78] and implemented for example in the MOMENT [11] project. In [14], a formal semantics was introduced for CBabel (i.e. a software architecture description languages (ADLs)) in rewriting logic using Maude. This is followed by formally verifying the producer-consumer-buffer problem using model checking and state search. In [78], a formal approach for the definition and analysis of domain-specific modelling languages was introduced based on standard model-driven engineering artifacts for defining a language's syntax (using metamodels) and its operational semantics (using model transformations) by translating them to the Maude. Therefore, meta-models and models are mapped to equational specifications, and model transformations are mapped to rewrite rules between such specifications due to Maude's reflective capabilities [20].

A number of interesting puzzles have been modelled and solved using rewriting logic language, Maude, in Chapter 7 in [20]. Moreover, Maude is used to solve the sudoku puzzle[1] in [82] using Maude strategies, where elimination was the main strategy and three processes for scanning, marking up, and analysis as the classical techniques for solving sudoku were followed. For a comprehensive list of rewriting logic foundations, logical and semantic framework, languages, tools and applications, we forward the reader to [54].

6.3 BP Compliance Problem

As for compliance problem, many approaches have been proposed for BP compliance [80] either enforcing the models to be compliant by design [89], or checking if designed model behaves as expected through the use of event logs and audits [94, 47]. A semi-automatic framework for managing compliance requirements and ensuring compliance throughout the BP lifecycle was introduced in [89]. In [47], a metamodel compliance

[1] http://en.wikipedia.org/wiki/Sudoku

checking was implemented in the tool Requirements Engineering Through Hypertext (RETH) which supports an object-oriented hypertext representation of requirements. Process mining techniques are also applied on process event logs and real-time data to monitor the behaviour of processes [94]. BPMN is used as system specifications in [6] where BPMN-Q, a graph-based query language based on BPMN, is introduced, used and integrated with Oryx, a web-based graphical modelling tool and repository, to model-check business rules compliance in PLTL.

To the best of our knowledge, the formal research on checking the compliance of BP models with CMMI process areas is still not enough to fully automate the procedure (e.g. [24, 23]). An interesting work in [24] investigates the relation between software quality models and businesses applying Model-Driven Development (MDD) using goal-oriented software approach. The degree of compliance of an industrially applied MDD approach with the CMMI-DEV quality model is analysed by determining the characteristics that meet the technical solution process area of CMMI-DEV and identify improvement opportunities to obtain a proper alignment of the MDD approach with the model. The check is done based on SCAMPI assessment evidences; affirmations (e.g. expert statements) and artifcats (e.g. tangible evidences). The same idea was applied to the requirement engineering process area in the CMMI-DEV in [23]. In [62], a formal compliance checking approach is introduced using Z notation for systems specifications and LTL for properties and NuSMV for model checking based on common criteria. Applying this approach to the international security standard (ISO/IEC 15408), a tool support called FORVEST was proposed in [106]. It provides information for the modeller on the formalization languages and tools to make the approach more accessible for business people.

There are many classifications for the compliance checking in literature. Figure 6.1 gives an idea about the classification of BPs compliance checking approaches. The reader can recall Table 6.2 for a summary comparison of most of the related compliance checking approaches. We provide the table below as Table 6.2. Generally, automated compliance checking has two approaches; forward and backward compliance checking [80, 34]. In forward compliance checking approaches, the compliance rules are verified during design time or execution time of the process allowing only for the complaint

behaviour to be carried out later [34]. Forward compliance checking approaches aims at preventing the non-compliant behaviour from occurring in the process execution. In backward compliance checking approaches, the check is used to detect that some non-compliant behaviour has occurred by checking the business process execution history (e.g. event logs). While forward compliance checking approaches can prevent non-complaint behaviour form happening, the backward compliance checking approaches cannot.

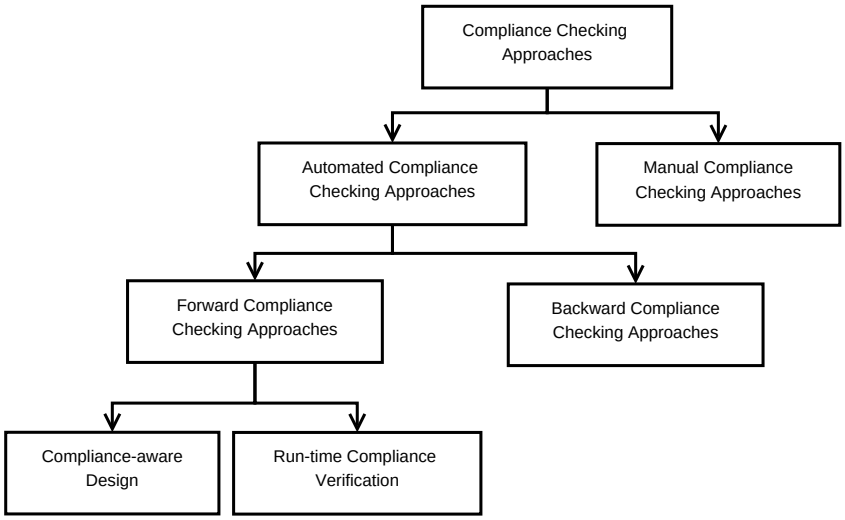


Figure 6.1: Compliance Checking Approaches Classification

There are two types of forward compliance checking approaches; compliance-aware design and run-time compliance checking. Compliance-aware design approaches work at design time; i.e. before the process is actually being deployed in real projects. The majority of recent approaches lies in this category such as [39, 51, 33, 41, 79, 84, 89] and it provides larger space for automation in defining the models and the compliance requirements. Moreover, the implementation of this approach is less expensive as the process is examined at the design-time. In design-time compliance checking approaches, either the design process is guided by the compliance requirements or the model checkers are used to verify that certain properties have been designed. An approach based on defining and using compliance patterns to compute the deviation of a given business process model to a certain compliance pattern was introduced in [39]. Based on control patterns, a formal definition of the compliance checking was presented in [65] and in [33], a semantic layer was added to the business process management stack in which process

instances are interpreted based on pre-defined set of controls. The work in [51] presents a support method which allows the modeller to quantitatively measure the compliance degree of a given process model based on a set of control objectives. In [41], the authors proposed a formalization approach for the business contracts semantics as well as their violations using Formal Contract Language (FCL) based on deontic logic. Although the authors in [41] built their approach based on deontic logic, the work in [79] shows the need for stronger logics to formalize the modelling of compliance controls. Moreover, they emphasis on the importance of automating the compliance checking and its relative regulations semantics. The approach in [84] proposed a compliance ontology and integrate it into the BP models. A comprehensive framework for semi-automatically managing compliance requirements which ensures compliance throughout all the phases of BP lifecycle was introduced in [89].

On the specification of compliance requirements, [79] proposes an approach for modelling control objectives within BP structures. Their work introduced a basic model to capture compliance requirements. In order to realize what is *compliance-by-design*, BP models are enriched with control tags. They propose a modal logic based approach using FCL, which separates the prescriptive modelling of processes and the descriptive nature of compliance requirements. However, the complexity of the adopted formal language poses critical problems in practice. In [38], the authors provides a forward design compliance checking approach. The approach is basically uses process models in EPCs which is mapped into Petri nets to check their compliance degree and maturity with an adopted model of the ITIL reference model on a designed plug in on ProM. The authors discussed the difference between process equivalence and process compliance as two process models can be compliant and not equivalent at the same time. That is the idea of having a general reference model which can be interpreted in many process models customized to specific business needs.

Run-time compliance checking approaches deal with executable business process models. In this case, compliance requirements can be defined into the business process models (as control flows) or can be dependent on a run-time information (as user or process input). The need to separate compliance modelling from process modelling is identified in [49], where business process models in BPEL are transformed into π-

Table 6.2: Summary comparison of some compliance checking approaches

Ref.	**System Spec**	**Property Spec**	**Check Proc.**	**Class.**	**App. Domain**	**Automation**
[6]	BPMN 2 PetriNets	BPMN-Q 2 PLTL	MC	FD	Banking	Oryx, Lola
[33]	BPEL	FCL 2 LTL	MC	F,B	Sarbens-Oxley Act	COMPAS
[94]	MXML	LTL	MC	B	event-logs	ProM
[24]	**MDD**	**BPRE4OO**	**SCAMPI**	**FDT**	**CMMI-DEV**	**NA**
[38]	EPC 2 PetriNets	ITIL adopted models	MC	FD	ITIL	ProM
[106]	Z	LTL	MC	F	ISO/IEC 15408	FORVEST
[49]	BPEL 2 π-calculus	BPSL 2 LTL	MC	FR	ITIL,COBIT	OPAL

Legend: D: Design-time, R: Run-time, MC: Model Checking, F: Forward, B: Backward, NA: Not Available, PM: Process Mining

calculus and compliance rules into graphical Business Property Specification Language (BPSL) are translated into LTL, then model checked. In [64], an approach is proposed to ensure effectiveness of controls during business process execution and a reaction strategy was designed in case of rules violation. The authors in [73] defined the BP models in a declarative way and argued that constraint-based workflow models are more expressive and flexible than procedural ones. The compliance policy definitions are integrated into the BP models in [59]. They are modelled within the process model events and transactions to monitor run-time compliance. This raises the need for formal definition of some of the important BPMN constructs, i.e. events, event triggers, and related resources, event patterns, message handling as well as state management [80].

Backward compliance checking aims at verifying that a business process execution is compliant with certain requirements and rules. In [77], the authors proposed a conformance checking technique to decide on how much a business process behaviour is similar to registered in process instances in a certain history log. The approach indicates where the differences exist using the business process graphical representation, though it did not handle data fields or user inputs. Another approach introduced in [94] where an LTL checker is introduced to check if an LTL formula holds for a certain process instance with references to the rule information sources. The approach is formal and lacks graphical representation which make it inaccessible for business users. In [15] and [5] the authors combines the power of formality and graphical rules representation using GOSpeL for graphical rule representation and translated into SCIFF (i.e. a declarative language based on computational logic) in order to produce process instances.

Manual approaches are traditionally used after running the process; i.e. retrospective reporting [42], by reviewing the resulting audits and logs. This approach largely depends on manual checks done by experts to figure out the non-compliant business process behaviour (violations). A small area of automation can be used in this case and turns the approach to be a backward automated approach by using data mining techniques [94] to detect violations from workflow logs using temporal logic. The authors in [94] used process mining techniques to monitor the behaviour of processes by using process event logs and real-time data. Possible deviations with process definition and compliance requirements are then detected and resolved.

The proposed compliance checking approach is considered a design-time forward automated compliance checking approach. The `W-BPMN` models are checked using model checking technique for compliance with CMMI-CM properties encoded in LTL.

6.4 Chapter Summary

In this chapter, the related work is presented in different related areas: BPMN formalization and verification, Maude applications for BPs, and BPs compliance checking. There are different categories under which the compliance check is studied and used to provide practical solutions for processes quality questions. The semantics in this work differs from the discussed related work with respect to the underlying logic (rewriting logic), and formal language (Maude), and the prospective use of the semantics in the application domain (compliance checking). The approach uses the LTL for property specifications of pattern-based requirements because Maude supports LTL and has its own LTL model checker which motivates the application side of the thesis.

Chapter 7

Conclusions and Future Work

This chapter concludes the thesis of compliance checking of well-formed BPMN models using Maude. In Section 7.1 we represent a summary of the work proposed in the thesis, Section 7.2 presents the conclusions of the work linking them with the original hypothesis in Chapter 1, while in Section 7.3 the set of approach limitations are discussed. Finally, in Section 7.4, some of the future research opportunities related to the work in this thesis are mentioned.

7.1 Summary

In this thesis, a semi-automated pre-appraisal approach for CMMI-CM compliance checking with formal BPMN models is proposed. The compliance checking is modelled as a model checking problem where the system specification is represented by the well-formed BPMN models in Maude and the property specifications is represented by the LTL properties mapped from the CMMI-CM compliance pattern based requirements. First, we present the formal syntax and behavioural semantics for a subset of the BPMN. The syntax is obtained by mapping the graphical elements in the BPMN into terms using the term rewriting system Maude. A number of BPMN challenging issues related to its ambiguity specifications and possibilities of deadlock and lack of synchronization models has been discussed. Moreover, a set of domain-specific rules are introduced for simulating the behaviour of BPMN data objects as the process essential resources using Maude (possibly conditional) rules. Unlike most of the BPMN formalizations, the proposed formalization defines a CFG for the guards expressions for decision based gateways in the gateways and not in the outgoing sequence flow, in order to allow for a decision to be made before considering any outgoing sequence flows. A comprehensive semantics for inclusive decision-based (OR) gateways is introduced by using bock structure. We introduced the well-formed BPMN model definition, which introduces design restrictions reducing the possibility of deadlocks. The function `wfs` is defined to check the consistency with well-formedness conditions in a BPMN model. Moreover, the formalization has proved to be sound based on the classical soundness definition. Second, the CMMI-CM requirements are formally represented into LTL properties. They mapped into LTL properties through the compliance patterns. The LTL properties are model checked using Maude LTL model checker against well-formed BPMN models one by one. In order to interpret the results of mode checking into meaningful compliance checking related results, we introduce a satisfaction grading scheme based on the number of sub-practices represented by the property.

7.2 Conclusions

Referring to the hypotheses introduced in Chapter 1, we conclude the thesis based on the results and discussions proposed throughout the thesis. In this section, the hypotheses are listed with the related contributions and references to the chapters.

1. ***What formalization of BPMN models can be considered suitable for compliance checking? If there is no such formalization, what are the main characteristics of a candidate formalization? Which formal language to use?***

   The existing BPs formalizations lacks essential constructs which are necessary for compliance checking problem, e.g. data objects (c.f. the comparison in Table 6.1 in Chapter 6). Therefore a new formalization is developed for an excerpt of the BPMN elements using Maude in Chapter 3 (e.g. activities, gateways, events, data objects, swimlanes, connecting flows). Data objects are given formal semantic behaviour allowing for status change and dependency relationships through domain-specific rules. A CFG and evaluation mechanism are proposed for the BPMN decision-based gateways which allow for more specific behaviour specifications. Moreover, the mapping from BPMN into Maude is verified and the semantics is proved to produce sound BP models to ensure that they are free of deadlocks and lack of synchronization in Chapter 4. In order to avoid possible structural errors in the formalized BPMN models, we introduced the notion of well-formed BPMN models, where the model is parsed for elements that are not following the syntactic constraints specified in Definition 3.3.2 for well-fromed BPMN models besides the standard description in the BPMN [68]. An introduction of Maude, as a formalization language, is in Chapter 2 and our Maude modules for BPMN syntax and semantics are presented in the attached code files (See Appendix A for details) and explained in Chapter 3. Due to the availability of Maude LTL model checker as part of Maude's verification toolkit, a model checking procedure is used to perform the compliance checking in Chapter 5 where Maude representation for the BPs is the model being checked against a set of LTL properties which represent the CMMI-CM requirements.

2. ***What are the characteristics of the CMMI process improvement model that make it an interesting area for compliance checking? Is it possible to formally represent the CMMI requirements? How?***

   The CMMI [21] process improvement model is used as the source of compliance requirements in this thesis. We believe this is the first formal representation for CMMI-CM requirements into LTL properties. CMMI is designed for software SMEs and its appraisal methods are time, effort and money consuming. The CM process area was chosen as an example for the reasons mentioned in in Section 1.4. A brief idea about the CMMI is introduced in Section 2.3 in Chapter 2. A comparison among the CMMI appraisals and our pre-appraisal approach is discussed into Section 2.3.2 in Chapter 2. We agreed with [6, 33, 96] that the LTL is suitable for the property specifications. The LTL is introduced in Section 5.2.1. On formalizing the CMMI requirements, they had to be mapped into compliance patterns based expressions and then mapped into the LTL properties (in Section 5.2.3) which are model checked in Section in 5.3 in Chapter 5.

3. ***What is the verification technique to formally check compliance of BP models with formal requirements? Is it able to provide an explicit answer to the question: "Is an input process compliant with the input set of properties?"?***

   The verification technique used in this thesis and explained in Chapter 5 is the model checking. The main reasons for that choice are listed in Section 1.4. As the model checkers normally returns a YES/NO answers (i.e. $true$ or $counterexample$), the output of the model checker in our approach had to be analysed to check the validity and real representation of the BP on hand. We have proposed a mechanism to check the existence of certain documents or objects in the model before taking it to be model checked in Section 5.3.2 in order to avoid spurious results. Moreover, a compliance checking grading scheme is proposed to interpret the model checking results.

4. ***What are the automation possibilities of the compliance checking approach?***

   The possibility of automating the compliance checking approach is promising as it allows the business people to be able to efficiently use the semi-automated approach. What encourages the automation of the approach is the textual compliance

requirements which can be mapped into compliance patterns and then into LTL properties. An overview of the automation of the approach is discussed in Section 7.4 in Chapter 7.

Recalling the main thesis statement, ***Is it possible to formally check the BPMN models compliance against the CMMI Configuration Management requirements?***, we can provide an answer to it. Yes, it is possible through implementing a formal pre-appraisal compliance checking approach using Maude as a formalization language for the system specifications and LTL for property specifications of the CMMI-CM requirements. Using Maude LTL model checker we propose a possible formal solution to ensure that a designed configuration management business process is compliant with CMMI-CM process area requirements as shown in this thesis. Table 7.1, which is recalled from Chapter 2 with adding our approach to it, shows a summary comparison between the appraisal methods and our approach. The proposed approach is formal and uses a customized automated model checking based on Maude LTL model checker to assess the business process compliance checking.

7.3 Limitations

Although we have tried our best to work on producing a comprehensive approach and method to add the formality to the compliance checking problem, there are number of drawbacks and limitations which the reader should be aware of. These limitations are listed below.

1. At the moment, a prior knowledge of some formal techniques and tools is required in order to apply the approach in real life cases. That is, to use the method, verifiers have to be familiar with BPMN, Maude, LTL, model checking, and CMMI.

2. The use of model checking as a verification tool has its own drawbacks which may affect the results of the approach, i.e. state space expansion [19]. Abstraction techniques [9, 20] may be used to solve the state space expansion in model checkers. However, we believe that the BPMN model for the configuration management process in a software company is of relatively reasonable size which will not normally cause the number of states to be a problem.

Table 7.1: Characteristics of CMMI appraisals and our approach

Feature	**Class A**	**Class B**	**Class C**	**Our Approach**
Usage Mode	In-depth investigation Basis for improvement	Self appraisal	Quick-look	Internal check for the Designed Process
Advantages	Strengths and Weaknesses of PAs Robust method with Consistent, repeatable results	A starting point focuses on areas that need most attention	Inexpensive, rapid feedback Short duration	Inexpensive, semi-automated Formal
Disadvantages	Demands significant resources	Not used for rating No deep coverage	Not used for rating Less ownership of results	Maude and BPMN are required Designed process only
Sponsor	Senior Manager	Any Manager	Any Internal Manager	Process Modeller
Team Size	4-10 and ATL[a]	1-6 and ATL	1-2 and ATL	1
Team Composition	External and internal	External or internal	External or internal	Internal

[a] ATL: Appraisal Team Leader.

3. The approach is not fully automated, however, the core functionality of the Maude-based formalization is fully provided. That is, the models can be verified to be well-formed, and therefore sound[1], model checked against the LTL properties of CMMI requirements automatically[2]. Other possible automation points are left as future work and explained in Section 7.4.

7.4 Future Work

The proposed approach can be extended in many ways; one is on the BPMN formalization side and the other is on property (CMMI requirements) formalization side.

(***BPMN Collaboration Models***) The formalization can be extended to model more BPMN elements for modelling the collaboration between more than one participants. The BP collaboration requires communications among different parties in the same company or in different organizations. In BPMN, such situations are modelled using swimlanes and message transfer. Although we presented a brief syntax for representing swimlanes and messages, a comprehensive semantics for the possible cases is still to be completed based on the proposed syntax and semantics in Chapter 3.

(***More BPMN Events***) An interesting family of BPMN constructs is the events, as BPMN has twelve different events, e.g. exception, cancel, and timer. Defining the possible errors or unexpected situations in design phase of a BP is essential to guarantee its accessibility and sustainability in all situations. The events can be part of a subprocess boundary event affecting the flow inside the subprocess or an intermediate event affecting its predecessor and successor elements. The basic constructs are already included into the proposed semantics, however, the formalization covers only the plain start, and end events, the intermediate error (exception), and messages.

(***More BPMN Gateways***) The formalization provided for the BPMN semantics in Chapter 3 is applicable to parallel AND, decision-based XOR, and decision-based OR gateways while the BPMN has also complex and event-based gateways which are essential for applications which use different events to control the flow. This point of enhancement in more related to the above point where the development of comprehensive

[1] See Chapter 4 for details.

[2] See Chapter 5 for details.

events semantics will aid the formalization of event-based gateways.

(***Bisimulation***) There is a study on progress to investigate the bisimulation (possibly weak) relation between the proposed semantics and the Petri net based formalization of BPMN in [27]. A well-formed BPMN model `W-BPMN` is mapped into a transition system and produced the possible traces of the state transitions. The workflow models in classical Petri nets have been formalized as a transition system in [90]. However, our semantics is not identically following the Petri nets semantics, as some rules defining the behaviour of gateways are considered as one step while there is one or more transitions on the other side in the Petri net model, e.g. AND fork. Moreover, the state definition in our semantics depends on the concept of activation, i.e. at state S an object o is inactive and then at state S' the same object is active, while in Petri net based formalizations, the states are the markings where the objects are transitions and the markings are only marking the places and not the transitions (i.e. transition is enabled if each of it its all input places have at least one token, and then it is fired after it is enabled). Therefore, a detailed investigation is still needed to decide if the Petri net formalization is suitable to be simulated with the proposed semantics, and if so, how the difference in the state definitions may affect the simulation relation.

(***Tool Support***) The current proposed approach provides the formalization of BPMN models in Maude and we are planning its integration within a modelling environment (e.g. Eclipse) using the mapping introduced in Chapter 3 of BPMN elements into Maude objects according to the proposed semantics to allow for automatic verification using Maude verification toolkit. A potential design for the tool support is given in Figure 7.1. Automation point (1) in the figure represents the mapping from the BPMN elements into Maude objects, while automation point (2) represents the mapping from the compliance pattern based requirements into the LTL formulae [33], and automation point (3) represent the option to automatically choose properties to model check them.

(***More CMMI***) The approach is still applicable for different process areas of the CMMI. Based on the fact that the appraisals are looking into the availability of evidences that certain components are part of the process and that they are used effectively, then the proposed approach allows the company to know which documents are part of their BP and which are not (using function `CE`) and then checks if these documents are used

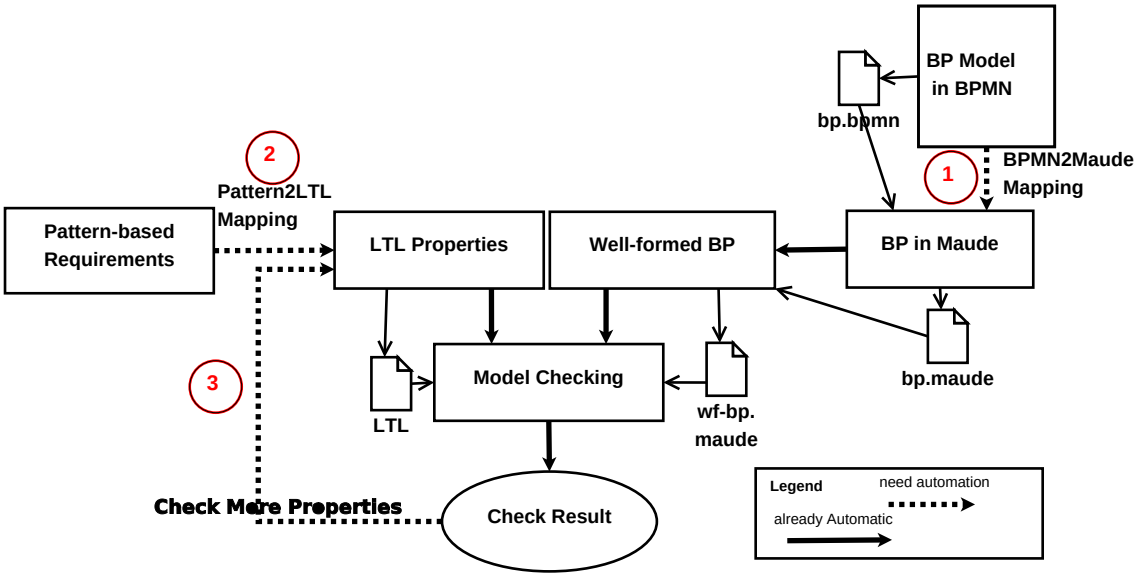


Figure 7.1: Potential Tool Support Design

properly (i.e. in a compliant way with the requirements) in the process through checking the related temporal properties. In the staged representation of the CMMI, the process area requirements in ML2 can be formalized using the compliance patterns and then mapped into LTL formulae and model checked against the corresponding BP models to decide on the company or project compliance with the CMMI. It adds the formality to the appraisals used for compliance checking in SMEs.

(***Real Scenario***) Applying the approach to a real life scenario was not possible during the development of this approach. However, the assessment of the approach with a real scenario is one of the essential future work to place the approach and enhance it to match the real scenarios features.

Bibliography

[1] W. M. P. Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In W. M. P. Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management*, volume 1806 of *LNCS*, pages 161–183. Springer Berlin Heidelberg, 2000.

[2] S. Abramsky, S. Gay, and R. Nagarajan. A Specification Structure for Deadlock-Freedom of Synchronous Processes. *Theoretical Computer Science*, 222(1-2):1–53, 1999.

[3] Y. K. Agarwal, B. Cary, S. Cash, L. Cassa, B. Demartini, C. Duplantis, A. N. de Godoi, D. B. Gomes, V. Gucer, M. Kipel, A. O. Neto, C. Saad, G. Shah, P. D. Tamarindo, and K. Venkitasubramanian. IBM Tivoli Change and Configuration Management Database (CCMDB) V7.2.1 Implementation Guide. Technical Report SG24-7879-00, IBM, 2010.

[4] D. M. Ahern, J. Armstrong, A. Clouse, J. R. Ferguson, and W. H. K. E. Nidiffer. *CMMI SCAMPI Distilled: Appraisals for Process Improvement*. Addison-Wesley Professional, 2005.

[5] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Expressing and Verifying Business Contracts with Abductive Logic Programming. *Int. J. Electron. Commerce*, 12(4):9–38, July 2008.

[6] A. Awad. *A Compliance Management Framework for Business Process Models*. PhD thesis, Hasso-Plattner-Institute, Potsdam, Germany, may 2010.

[7] A. Awad and F. Puhlmann. Structural Detection of Deadlocks in Business Process

Models. In W. Abramowicz and D. Fensel, editors, *BIS*, volume 7 of *LNBIP*, pages 239–250. Springer, 2008.

[8] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. Wijngaarden, and M. Woodger. Report on the Algorithmic Language ALGOL 60. *Numerische Mathematik*, 2(1):106–136, 1960.

[9] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[10] E. Börger. Approaches to Modeling Business Processes: a Critical Analysis of BPMN, Workflow Patterns and YAWL. *Software & Systems Modeling*, 11(3):305–318, 2012.

[11] A. Boronat. *MOMENT: A Formal Framework for MOdel managemMENT*. PhD thesis, Universitat Politécnica de Valéncia, June 2007.

[12] A. Boronat and J. Meseguer. Automated Model Synchronization: A Case Study on UML with Maude. *Electronic Communications of the EASST, Graph Transformation and Visual Modeling Techniques*, 41, 2011.

[13] K. Boukhelfa, F. Belala, A. Choutri, and H. Douibi. For More Understandable UML Diagrams. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, AICCSA'10, pages 1–7. IEEE Computer Society, 2010.

[14] C. Braga and A. Sztajnberg. Towards a Rewriting Semantics for a Software Architecture Description Language. *ENTCS*, 95(0):149–168, 2004. Proceedings of the Brazilian Workshop on Formal Methods.

[15] F. Chesani, P. Mello, M. Montali, and S. Storari. Testing Careflow Process Execution Conformance by Translating a Graphical Language to Computational Logic. In R. Bellazzi, A. Abu-Hanna, and J. Hunter, editors, *Artificial Intelligence in Medicine*, volume 4594 of *LNCS*, pages 479–488. Springer Berlin Heidelberg, 2007.

[16] I. M. Chiswell. Context-free Languages. In *A Course in Formal Languages, Automata and Groups*, Universitext, pages 1–33. Springer London, 2009.

[17] N. Chomsky. Three Models for The Description of Language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[18] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In D. Kozen, editor, *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[19] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model Checking and the State Explosion Problem. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *LNCS*, pages 1–30. Springer Berlin Heidelberg, 2012.

[20] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer, 2007.

[21] CMU-SEI. CMMI for Development, Version 1.3. Technical Report CMU/SEI-2010-TR-033, Software Engineering Institute, Carnegie Mellon University, 2010.

[22] CMU-SEI. Capability Maturity Model Integration. Technical report, Carnegie Mellon University, Software Engineering Institution, 2011.

[23] A. M. L. de Vasconcelos, J. L. de la Vara, J. Sánchez, and O. Pastor. Towards CMMI-compliant Business Process-Driven Requirements Engineering. In J. P. Faria, A. R. da Silva, and R. J. Machado, editors, *QUATIC*, pages 193–198. IEEE Computer Society, 2012.

[24] A. M. L. de Vasconcelos, G. Giachetti, B. Marín, and O. Pastor. Towards a CMMI-Compliant Goal-Oriented Software Process through Model-Driven Development. In P. Johannesson, J. Krogstie, and A. L. Opdahl, editors, *PoEM*, volume 92 of *LNBIP*, pages 253–267. Springer, 2011.

[25] J. Dehnert and P. Rittgen. Relaxed Soundness of Business Processes. In K. R. Dittrich, A. Geppert, and M. C. Norrie, editors, *Advanced Information Systems*

Engineering, volume 2068 of *LNCS*, pages 157–170. Springer Berlin Heidelberg, 2001.

[26] G. Denker, J. Meseguer, and C. Talcott. Formal Specification and Analysis of Active Networks and Communication Protocols: the Maude Experience. In *Proc. of DISCEX'00*, volume 1, pages 251–265, 2000.

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

[28] R. M. Dijkman and P. V. Gorp. BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules. In J. Mendling, M. Weidlich, and M. Weske, editors, *Business Process Modeling Notation - Second International Workshop, BPMN 2010*, volume 67 of *LNBIP*, pages 16–30. Springer, 2010.

[29] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[30] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In F. Gaducci and U. Montanari, editors, *Proceedings of WRLA 2002*, volume 71 of *ENTCS*, Amsterdam, September 2002. Elsevier.

[31] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker and its Implementation. In *In Model Checking Software: Proc. 10 th Intl. SPIN Workshop*, pages 230–234. Springer LNCS, 2003.

[32] A. Elgammal. *Towards A Comprehensive Framework for Business Process Compliance*. PhD thesis, Tilburg University, April 2012.

[33] A. Elgammal, O. Turetken, W.-J. Heuvel, and M. Papazoglou. Formalizing and Appling Compliance Patterns for Business Process Compliance. *Software & Systems Modeling*, pages 1–28, 2014.

[34] M. ElKharbili, A. K. A. de Medeiros, S. Stein, and W. M. P. van der Aalst. Business Process Compliance Checking: Current State and Future Challenges. In P. Loos, M. Nüttgens, K. Turowski, and D. Werth, editors, *MobIS*, volume 141 of *LNI*, pages 107–113. GI, 2008.

[35] H. Endert, B. Hirsch, T. Küster, and S. Albayrak. Towards a Mapping from BPMN to Agents. In *Proceedings of the 2007 AAMAS international workshop and SOCASE 2007 conference on Service-oriented computing: agents, semantics, and engineering*, AAMAS'07/SOCASE'07, pages 92–106. Springer-Verlag, 2007.

[36] A. Farzan and J. Meseguer. State Space Reduction of Rewrite Theories Using Invisible Transitions. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 142–157. Springer Berlin Heidelberg, 2006.

[37] S. Garcia-Miller. Thoughts on Applying CMMI in Small Settings. Technical report, Carnegie Mellon, SEI, 2005.

[38] K. Gerke, J. Cardoso, and A. Claus. Measuring the Compliance of Processes with Reference Models. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5870 of *LNCS*, pages 76–93. Springer Berlin Heidelberg, 2009.

[39] A. Ghose and G. Koliadis. Auditing Business Process Compliance. In B. Krämer, K.-J. Lin, and P. Narasimhan, editors, *Service-Oriented Computing, ICSOC 2007*, volume 4749 of *LNCS*, chapter 14, pages 169–180. Springer Berlin, Heidelberg, 2007.

[40] P. V. Gorp and R. M. Dijkman. A Visual token-based Formalization of BPMN 2.0 based on in-place Transformations. *Information & Software Technology*, 55(2):365–394, 2013.

[41] G. Governatori, Z. Milosevic, and S. Sadiq. Compliance Checking Between Business Processes and Business Contracts. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '06, pages 221–232, Washington, DC, USA, 2006. IEEE Computer Society.

[42] G. Governatori and S. Sadiq. The Journey to Business Process Compliance. *Handbook of Research on BPM*, pages 426–454, 2009.

[43] L. H. Grande. Introducción a la notación BPMN y su relación con las estrategias del lenguaje Maude. Master's thesis, Universidad Complutense de Madrid, 2009.

[44] K. Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In W. M. P. Aalst and E. Best, editors, *Applications and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 337–356. Springer Berlin Heidelberg, 2003.

[45] ISO. ISO 10007:2003 Quality Management Systems Guidelines for Configuration Management. Accessed online in 06-03-2014.

[46] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[47] K. S. H. M. H. V. Kaindl, H. Metamodel-Compliance Checking of Requirements in a Semiformal Representation. In *the 15th Conference on Advanced Information Systems Engineering*, volume 74 of *CAiSE '03*, 2003.

[48] A. Ligeza, K. Kluza, and T. Potempa. AI Approach to Formal Analysis of BPMN Models. Towards a Logical Model for BPMN Diagrams. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 931–934, 2012.

[49] Y. Liu, S. Müller, and K. Xu. A Static Compliance-Checking Framework for Business Process Models. *IBM Systems Journal*, 46(2):335–361, Apr. 2007.

[50] J. O. Long. *ITIL Version 3 at a Glance Information Quick Reference*. Springer-Verlag US, 2008.

[51] R. Lu, S. Sadiq, and G. Governatori. Compliance Aware Business Process Design. In *Proceedings of the 2007 international conference on Business Process Management*, BPM'07, pages 120–131, Berlin, Heidelberg, 2008. Springer-Verlag.

[52] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[53] A. Martens. On Compatibility of Web Services. *Petri Net Newsletter*, 65:12–20, 2003.

[54] N. MartíOliet, M. Palomino, and A. Verdejo. Rewriting Logic Bibliography by Topic: 1990-2011. *The Journal of Logic and Algebraic Programming*, 81(7-8):782–815, 2012. Rewriting Logic and its Applications.

[55] J. Mendling. *Detection and Prediction of Errors in EPC Business Process Models*. PhD thesis, Institute of Information Systems and New Media, Vienna University of Economics and Business Administration, May 2007.

[56] J. Meseguer. Conditional Rewriting Logic As a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, Apr. 1992.

[57] J. Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: a Progress Report. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer Berlin Heidelberg, 1996.

[58] J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *the 12th International Workshop on Recent Trends in Algebraic Development Techniques Proceedings*, WADT'97, pages 18–61. Springer-Verlag, 1997.

[59] Z. Milosevic. Towards Integrating Business Policies with Business Processes. In *Proceedings of the 3rd international conference on Business Process Management*, BPM'05, pages 404–409, Berlin, Heidelberg, 2005. Springer-Verlag.

[60] I. Minnich. CMMI Appraisal Methodologies: Choosing What Is Right for You. *CROSSTALK, The Journal of Defense Software Engineering*, 15(2):7–8, 2002.

[61] F. Mokhati, P. Gagnon, and M. Badri. Verifying UML Diagrams with Model Checking: A Rewriting Logic Based Approach. In *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pages 356–362, Oct 2007.

[62] S. Morimoto, S. Shigematsu, Y. Goto, and J. Cheng. Formal Verification of Security Specifications with Common Criteria. In *Proceedings of the 2007 ACM*

Symposium on Applied Computing, SAC '07, pages 1506–1512, New York, NY, USA, 2007. ACM.

[63] M. Muehlen and J. Recker. How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In Z. Bellahséne and M. Léonard, editors, *Advanced Information Systems Engineering*, volume 5074 of *LNCS*, pages 465–479. Springer, 2008.

[64] K. Namiri and N. Stojanovic. Pattern-based Design and Validation of Business Process Compliance. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM'07, pages 59–76, Berlin, Heidelberg, 2007. Springer-Verlag.

[65] K. Namiri and N. Stojanovic. Towards A Formal Framework for Business Process Compliance. In M. Bichler, T. Hess, H. Krcmar, U. Lechner, F. Matthes, A. Picot, B. Speitkamp, and P. Wolf, editors, *Multikonferenz Wirtschaftsinformatik*. GITO-Verlag, Berlin, 2008.

[66] R. S. Nandyal. *Making Sense of Software Quality Assurance*. TBS, 2008.

[67] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

[68] OMG. Business Process Model and Notation (BPMN) Version 2.0. Technical Report formal/2011-01-03, OMG, 2011.

[69] S. Onoda, Y. Ikkai, T. Kobayashi, and N. Komoda. Definition of Deadlock Patterns for Business Processes Workflow Models. In *Systems Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*, volume Track 5, pages 1–11, 1999.

[70] C. Ouyang, M. Dumas, S. Breutel, and A. ter Hofstede. Translating Standard Process Models to BPEL. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering*, CAiSE'06, pages 417–432. Springer-Verlag, 2006.

[71] R. Pelánek. Fighting State Space Explosion: Review and Evaluation. In D. Cofer and A. Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *LNCS*, pages 37–52. Springer Berlin Heidelberg, 2009.

[72] J. R. Persse. *Process Improvement Essentials: CMMI, Six SIGMA, and ISO 9001*. Theory in Practice. O'Reilly Media, Inc., 2006.

[73] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van Der Aalst. Constraint-based Workflow Models: Change Made Easy. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM'07, pages 77–94, Berlin, Heidelberg, 2007. Springer-Verlag.

[74] F. Puhlmann and M. Weske. Using the π-calculus for Formalizing Workflow Patterns. In *Proceedings of the 3rd International Conference on Business Process Management*, BPM'05, pages 153–168. Springer-Verlag, 2005.

[75] F. Puhlmann and M. Weske. Investigations on Soundness Regarding Lazy Activities. In *Proceedings of the 4th international conference on Business Process Management*, BPM'06, pages 145–160. Springer-Verlag, 2006.

[76] W. Reisig. *Undersanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer-Verlag Berlin Heidelberg, 2013.

[77] A. Rozinat and W. M. P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. *Inf. Syst.*, 33(1):64–95, 2008.

[78] V. Rusu. Embedding Domain-Specific Modelling Languages in Maude Specifications. *Software & Systems Modeling*, 12(4):847–869, 2013.

[79] S. Sadiq, G. Governatori, and K. Namiri. Modeling Control Objectives for Business Process Compliance. In G. Alonso, P. Dadam, and M. Rosemann, editors, *Business Process Management*, volume 4714 of *LNCS*, pages 149–164. Springer-Verlag, 2007.

[80] S. W. Sadiq. A Roadmap for Research in Business Process Compliance. In W. Abramowicz, L. Maciaszek, and K. Wecel, editors, *BIS (Workshops)*, volume 97 of *LNBIP*, pages 1–4. Springer, 2011.

[81] W. Sadiq and M. E. Orlowska. Analyzing Process Models Using Graph Reduction Techniques. *Information Systems*, 25(2):117–134, 2000.

[82] G. Santos-García and M. Palomino. Solving Sudoku Puzzles with Rewriting Rules. *ENTCS*, 176(4):79–93, July 2007.

[83] SCAMPI-Upgrade-Team. Standard CMMI Appraisal Method for Process Improvement (SCAMPI) A, Version 1.3: Method Definition Document. Technical Report CMU/SEI-2011-HB-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2011.

[84] R. Schmidt, C. Bartsch, and R. Oberhauser. Ontology-based Representation of Compliance Requirements for Service Processes. In M. Hepp, K. Hinkelmann, D. Karagiannis, R. Klein, and N. Stojanovic, editors, *SBPM*, volume 251 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[85] I. C. Society. IEEE Standard for Configuration Management in Systems and Software Engineering. *IEEE Std 828âĎć-2012*, pages i–58, 2012.

[86] M. Staples and M. Niazi. Two Case Studies on Small Enterprise Motivation and Readiness for CMMI. In *Proceedings of the 11th International Conference on Product Focused Software*, PROFES '10, pages 63–66, New York, NY, USA, 2010. ACM.

[87] M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Rewriting Logic as a Unifying Framework for Petri Nets. In H. Ehrig, J. Padberg, G. Juhás, and G. Rozenberg, editors, *Unifying Petri Nets*, volume 2128 of *LNCS*, pages 250–303. Springer Berlin Heidelberg, 2001.

[88] P. Thati, K. Sen, and N. Martí-Oliet. An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0. *ENTCS*, 71:261–281, 2002.

[89] O. Türetken, A. Elgammal, W.-J. van den Heuvel, and M. P. Papazoglou. Enforcing Compliance on Business Processes through the use of Patterns. In V. K. Tuunainen, M. Rossi, and J. Nandhakumar, editors, *ECIS*, 2011.

[90] W. van der Aalst and C. Stahl. *Modeling Business Processes: A Petri Net-Oriented Approach*. Massachusetts Institute of Technology, 2011.

[91] W. M. van der Aalst and A. H. M. T. Hofstede. YAWL: Yet Another Workflow Language. *Information Systems Journal*, 30:245–275, 2003.

[92] W. M. P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *ICATPN*, volume 1248 of *LNCS*, pages 407–426. Springer, 1997.

[93] W. M. P. van der Aalst. Making Work Flow: On the Application of Petri Nets to Business Process Management. In J. Esparza and C. Lakos, editors, *ICATPN*, volume 2360 of *LNCS*, pages 1–22. Springer, 2002.

[94] W. M. P. van der Aalst, H. T. de Beer, and B. F. van Dongen. Process Mining and Verification of Properties: An Approach based on Temporal Logic. In *Proceedings of the 2005 Confederated International Conference on On the Move to Meaningful Internet Systems - Volume Part I*, OTM'05, pages 130–147, Berlin, Heidelberg, 2005. Springer-Verlag.

[95] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of Workflow Nets: Classification, Decidability, and Analysis. *Formal Aspe. Comput.*, 23(3):333–363, 2011.

[96] M. Y. Vardi. Branching vs. Linear Time: Final Showdown. In T. Margaria and W. Yi, editors, *TACAS*, volume 2031 of *LNCS*, pages 1–22. Springer, 2001.

[97] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[98] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. *Electr. Notes Theor. Comput. Sci.*, 71:282–300, 2002.

[99] A. Verdejo and N. Martí-Oliet. Two Case Studies of Semantics Execution in Maude: CCS and LOTOS. *Formal Methods in System Design*, 27(1-2):113–172, 2005.

[100] H. Völzer. A New Semantics for The Inclusive Converging Gateway in Safe Processes. In *Proceedings of the 8th International Conference on Business Process Management*, BPM'10, pages 294–309. Springer-Verlag, 2010.

[101] K. V. Wal. ISACA and IT Governance Institute Annual Report. `http://www.isaca.org/COBIT/Pages/default.aspx?cid=1003566&Appeal=PR`, 2011. Accessed online in 4-12-2013.

[102] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag Berlin Heidelberg, 2012.

[103] P. Wohed, W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, and N. Russell. Pattern-based Analysis of BPMN : An Extensive Evaluation of the Control-flow, the Data and the Resource Perspectives. *BPM Center Report BPM-05-26*, 2005.

[104] P. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM'08, pages 355–374. Springer-Verlag, 2008.

[105] P. Wong and J. Gibbons. Formalisations and Applications of BPMN. *Science of Computer Programming*, 76(8):633–650, Aug. 2011.

[106] K. Yajima, S. Morimoto, D. Horie, N. S. Azreen, Y. Goto, and J. Cheng. FORVEST: A Support Tool for Formal Verification of Security Specifications with ISO/IEC 15408. In *International Conference on Availability, Reliability and Security (ARES'09)*, pages 624–629, March 2009.

[107] J. Ye and W. Song. Transformation of bpmn diagrams to yawl nets. *Journal of Software*, 5(4):396–404, 2010.

Appendix A

Appendices List

The appendices of this thesis include the following:

1. CMMI Configuration Management PA. A copy of the CMMI CM process area guidelines is included in Appendix B.

2. Designed Maude Functions. In Appendix C, the functions used in the semantics and well-formed checking are defined and explained.

3. Maude Modules. The code files can be downloaded from the University of Leicester research archive at the same page the thesis downloaded from. The files includes modules (BPMN-SYNTAX, BPMN-SEMANTICS, WFS-BPMN, BPMN-EXAMPLES, CM-PREDS, CM-CHECK and Maude model checking modules). A file named (Counterexamples.txt) includes all the counterexamples we have when running the model checker for EX1 and EX2. Note that, the counterexample file is a collection of counterexamples and not readable by Maude. Moreover, a *read me* text file is giving some suggested steps to run the modules. Preliminary link to the file is: `https://lra.le.ac.uk/handle/2381/385`, then search by the thesis title or the author name.

Appendix B

CMMI-CM Process Area

Configuration Management is a support process area at Maturity Level 2 in CMMI. The following is the description of the process area in the CMMI-DEV 1.3 document by [21].

Purpose: The purpose of Configuration Management (CM) is to establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits.

Introductory Notes: The Configuration Management process area involves the following activities:

- Identifying the configuration of selected work products that compose baselines at given points in time
- Controlling changes to configuration items
- Building or providing specifications to build work products from the configuration management system
- Maintaining the integrity of baselines
- Providing accurate status and current configuration data to developers, end users, and customers

The work products placed under configuration management include the products that are delivered to the customer, designated internal work products, acquired products, tools, and other items used in creating and describing these work products.

Examples of work products that can be placed under configuration management include: Hardware and equipment, Drawings, Product specifications, Tool configurations, Code and li-

braries, Compilers, Test tools and test scripts, Installation logs, Product data files, Product technical publications, Plans, User stories, Iteration backlogs, Process descriptions, Requirements, Architecture documentation and design data, Product line plans, processes, and core assets.

Acquired products may need to be placed under configuration management by both the supplier and the project. Provisions for conducting configuration management should be established in supplier agreements. Methods to ensure that data are complete and consistent should be established and maintained. Configuration management of work products can be performed at several levels of granularity. Configuration items can be decomposed into configuration components and configuration units. Only the term *configuration item* is used in this process area. Therefore, in these practices, *configuration item* (CI) may be interpreted as configuration component or configuration unit as appropriate.

Baselines provide a stable basis for the continuing evolution of configuration items. An example of a baseline is an approved description of a product that includes internally consistent versions of requirements, requirement traceability matrices, design, discipline-specific items, and end-user documentation. Baselines are added to the configuration management system as they are developed. Changes to baselines and the release of work products built from the configuration management system are systematically controlled and monitored via the configuration control, change management, and configuration auditing functions of configuration management.

This process area applies not only to configuration management on projects but also to configuration management of organizational work products such as standards, procedures, reuse libraries, and other shared supporting assets. Configuration management is focused on the rigorous control of the managerial and technical aspects of work products, including the delivered product or service. This process area covers the practices for performing the configuration management function and is applicable to all work products that are placed under configuration management.

For product lines, configuration management involves additional considerations due to the sharing of core assets across the products in the product line and across multiple versions of core assets and products.

Specific Goal and Practice Summary

SG 1 Establish Baselines

SP 1.1 Identify Configuration Items

SP 1.2 Establish a Configuration Management System

SP 1.3 Create or Release Baselines

SG 2 Track and Control Changes

SP 2.1 Track Change Requests

SP 2.2 Control Configuration Items

SG 3 Establish Integrity

SP 3.1 Establish Configuration Management Records

SP 3.2 Perform Configuration Audits

B.1 SG 1 Establish Baselines

Baselines of identified work products are established. Specific practices to establish baselines are covered by this specific goal. The specific practices under the Track and Control Changes specific goal serve to maintain the baselines. The specific practices of the Establish Integrity specific goal document and audit the integrity of the baselines.

SP 1.1 Identify Configuration Items

Identify configuration items, components, and related work products to be placed under configuration management.

Configuration identification is the selection and specification of the following:

- Products delivered to the customer
- Designated internal work products
- Acquired products
- Tools and other capital assets of the project's work environment
- Other items used in creating and describing these work products

Configuration items (CIs) can include hardware, equipment, and tangible assets as well as software and documentation. Documentation can include requirements specifications and interface documents. Other documents that serve to identify the configuration of the product or service, such as test results, may also be included. A CI is an entity designated for configuration management, which may consist of multiple related work products that form a baseline. This logical grouping provides ease of identification and controlled access. The selection of work products

for configuration management should be based on criteria established during planning. Example Work Products is Identified configuration items.

Subpractices

(1.1.1) Select configuration items and work products that compose them based on documented criteria.

(1.1.2) Assign unique identifiers to configuration items.

(1.1.3) Specify the important characteristics of each configuration item.

(1.1.4) Specify when each configuration item is placed under configuration management.

(1.1.5) Identify the owner responsible for each configuration item.

(1.1.6) Specify relationships among configuration items.

Example criteria for selecting configuration items at the appropriate work product level include the following:

- Work products that can be used by two or more groups
- Work products that are expected to change over time either because of errors or changes in requirements
- Work products that are dependent on each other (i.e., a change in one mandates a change in the others)
- Work products critical to project success

Examples of work products that may be part of a configuration item include the following:

- Design
- Test plans and procedures
- Test results
- Interface descriptions
- Drawings
- Source code

- User stories or story cards
- The declared business case, logic, or value
- Tools (e.g., compilers)
- Process descriptions
- Requirements

Example characteristics of configuration items include author, document or file type, programming language for software code files, minimum marketable features, and the purpose the configuration item serves.

Example criteria for determining when to place work products under configuration management include the following:

- When the work product is ready for test
- Stage of the project lifecycle
- Degree of control desired on the work product
- Cost and schedule limitations
- Stakeholder requirements

Incorporating the types of relationships (e.g., parent-child, dependency) that exist among configuration items into the configuration management structure (e.g., configuration management database) assists in managing the effects and impacts of changes.

SP 1.2 Establish a Configuration Management System

Establish and maintain a configuration management and change management system for controlling work products.

A configuration management system includes the storage media, procedures, and tools for accessing the system. A configuration management system can consist of multiple subsystems with different implementations that are appropriate for each configuration management environment. A change management system includes the storage media, procedures, and tools for recording and accessing change requests.

Example Work Products

- Configuration management system with controlled work products
- Configuration management system access control procedures
- Change request database

Subpractices

(1.2.1) Establish a mechanism to manage multiple levels of control.

(1.2.2) Store and retrieve configuration items in a configuration management system.

(1.2.3) Provide access control to ensure authorized access to the configuration management system.

(1.2.4) Share and transfer configuration items between control levels in the configuration management system.

(1.2.5) Store and recover archived versions of configuration items.

(1.2.6) Store, update, and retrieve configuration management records.

(1.2.7) Create configuration management reports from the configuration management system.

(1.2.8) Preserve the contents of the configuration management system.

(1.2.9) Revise the configuration management structure as necessary.

The level of control is typically selected based on project objectives, risk, and resources. Control levels can vary in relation to the project lifecycle, type of system under development, and specific project requirements. Example levels of control include the following:

- Uncontrolled: Anyone can make changes.
- Work-in-progress: Authors control changes.
- Released: A designated authority authorizes and controls changes and relevant stakeholders are notified when changes are made.

Levels of control can range from informal control that simply tracks changes made when configuration items are being developed to formal configuration control using baselines that can only be changed as part of a formal configuration management process.

Examples of preservation functions of the configuration management system include the following:

- Backup and restoration of configuration management files
- Archive of configuration management files
- Recovery from configuration management errors

SP 1.3 Create or Release Baselines

Create or release baselines for internal use and for delivery to the customer.

A baseline is represented by the assignment of an identifier to a configuration item or a collection of configuration items and associated entities at a distinct point in time. As a product or service evolves, multiple baselines can be used to control development and testing. Hardware products as well as software and documentation should also be included in baselines for infrastructure related configurations (e.g., software, hardware) and in preparation for system tests that include interfacing hardware and software. One common set of baselines includes the system level requirements, system element level design requirements, and the product definition at the end of development/beginning of production. These baselines are typically referred to respectively as the *functional baseline*, *allocated baseline*, and *product baseline*.

A software baseline can be a set of requirements, design, source code files and the associated executable code, build files, and user documentation (associated entities) that have been assigned a unique identifier. Example work products are: baselines, description of baselines.

Subpractices

(1.3.1) Obtain authorization from the CCB before creating or releasing baselines of configuration items.

(1.3.2) Create or release baselines only from configuration items in the configuration management system.

(1.3.3) Document the set of configuration items that are contained in a baseline.

(1.3.4) Make the current set of baselines readily available.

B.2 SG 2 Track and Control Changes

Changes to the work products under configuration management are tracked and controlled. The specific practices under this specific goal serve to maintain baselines after they are established

by specific practices under the Establish Baselines specific goal.

SP 2.1 Track Change Requests

Track change requests for configuration items.

Change requests address not only new or changed requirements but also failures and defects in work products. Change requests are analyzed to determine the impact that the change will have on the work product, related work products, the budget, and the schedule. Example work products: Change requests.

Subpractices

(2.1.1) Initiate and record change requests in the change request database.

(2.1.2) Analyze the impact of changes and fixes proposed in change requests.

(2.1.3) Categorize and prioritize change requests.

(2.1.4) Review change requests to be addressed in the next baseline with relevant stakeholders and get their agreement.

(2.1.5) Track the status of change requests to closure.

Changes are evaluated through activities that ensure that they are consistent with all technical and project requirements. Changes are evaluated for their impact beyond immediate project or contract requirements. Changes to an item used in multiple products can resolve an immediate issue while causing a problem in other applications. Changes are evaluated for their impact on release plans.

Emergency requests are identified and referred to an emergency authority if appropriate. Changes are allocated to future baselines.

Conduct the change request review with appropriate participants. Record the disposition of each change request and the rationale for the decision, including success criteria, a brief action plan if appropriate, and needs met or unmet by the change. Perform the actions required in the disposition and report results to relevant stakeholders.

Change requests brought into the system should be handled in an efficient and timely manner. Once a change request has been processed, it is critical to close the request with the appropriate approved action as soon as it is practical. Actions left open result in larger than necessary status lists, which in turn result in added costs and confusion.

SP 2.2 Control Configuration Items

Control changes to configuration items.

Control is maintained over the configuration of the work product baseline. This control includes tracking the configuration of each configuration item, approving a new configuration if necessary, and updating the baseline. Example work products: Revision history of configuration items, and Archives of baselines.

Subpractices

(2.2.1) Control changes to configuration items throughout the life of the product or service.

(2.2.2) Obtain appropriate authorization before changed configuration items are entered into the configuration management system (e.g. authorization from the CCB, the project manager, product owner, or the customer).

(2.2.3) Check in and check out configuration items in the configuration management system for incorporation of changes in a manner that maintains the correctness and integrity of configuration items.

(2.2.4) Perform reviews to ensure that changes have not caused unintended effects on the baselines (e.g., ensure that changes have not compromised the safety or security of the system).

(2.2.5) Record changes to configuration items and reasons for changes as appropriate.

Examples of check-in and check-out steps include the following:

- Confirming that the revisions are authorized
- Updating the configuration items
- Archiving the replaced baseline and retrieving the new baseline
- Commenting on the changes made to the item
- Tying changes to related work products such as requirements, user stories, and tests

If a proposed change to the work product is accepted, a schedule is identified for incorporating the change into the work product and other affected areas. Configuration control mechanisms can be tailored to categories of changes. For example, the approval considerations could be less stringent for component changes that do not affect other components. Changed configuration

items are released after review and approval of configuration changes. Changes are not official until they are released.

B.3 SG 3 Establish Integrity

Integrity of baselines is established and maintained.

The integrity of baselines, established by processes associated with the Establish Baselines specific goal, and maintained by processes associated with the Track and Control Changes specific goal, is addressed by the specific practices under this specific goal.

SP 3.1 Establish Configuration Management Records

Establish and maintain records describing configuration items. Example work products:

- Revision history of configuration items
- Change log
- Change request records
- Status of configuration items
- Differences between baselines

Subpractices

(3.1.1) Record configuration management actions in sufficient detail so the content and status of each configuration item is known and previous versions can be recovered.

(3.1.2) Ensure that relevant stakeholders have access to and knowledge of the configuration status of configuration items.

(3.1.3) Identify the version of configuration items that constitute a particular baseline.

(3.1.4) Describe differences between successive baselines.

(3.1.5) Specify the latest version of baselines.

(3.1.6) Revise the status and history (i.e., changes, other actions) of each configuration item as necessary.

Examples of activities for communicating configuration status include the following:

- Providing access permissions to authorized end users
- Making baseline copies readily available to authorized end users
- Automatically alerting relevant stakeholders when items are checked in or out or changed, or of decisions made regarding change requests

SP 3.2 Perform Configuration Audits

Perform configuration audits to maintain the integrity of configuration baselines.

Configuration audits confirm that the resulting baselines and documentation conform to a specified standard or requirement. Configuration item related records can exist in multiple databases or configuration management systems. In such instances, configuration audits should extend to these other databases as appropriate to ensure accuracy, consistency, and completeness of configuration item information. Example work products are: Configuration audit results and Action items. Examples of audit types include the following:

- *Functional configuration audits (FCAs)*: Audits conducted to verify that the development of a configuration item has been completed satisfactorily, that the item has achieved the functional and quality attribute characteristics specified in the functional or allocated baseline, and that its operational and support documents are complete and satisfactory.
- *Physical configuration audits (PCAs)*: Audits conducted to verify that a configuration item, as built, conforms to the technical documentation that defines and describes it.
- *Configuration management audits*: Audits conducted to confirm that configuration management records and configuration items are complete, consistent, and accurate.

Subpractices

(3.2.1) Assess the integrity of baselines.

(3.2.2) Confirm that configuration management records correctly identify configuration items.

(3.2.3) Review the structure and integrity of items in the configuration management system.

(3.2.4) Confirm the completeness, correctness, and consistency of items in the configuration management system. Completeness, correctness, and consistency of the configuration management system's content are based on requirements as stated in the plan and the disposition of approved change requests.

(3.2.5) Confirm compliance with applicable configuration management standards and procedures.

(3.2.6) Track action items from the audit to closure

Appendix C

Maude Functions

In this appendix we present explanations of the semantics defined functions in Maude. In Section C.1, functions `preds`, `succs`, and `OinP` are defined for returning an object's predecessor, successors and pool name. Followed by the functions used to define the well-formed BPMN models in Section C.2. The function used to define the semantic rules for gateways are defined in Section C.3 for AND related functions and in Section C.4 for the decision gateway related functions.

C.1 Operations on Business Process Models

We present here some useful functions that operate on the BP models and are build upon the proposed formalization of BPMN syntax and semantics in Maude. Working on finding the paths forward and backward leads us to have equations that retrieve a set of *all* objects that are before/after a certain object (or its predecessors/successors). In case of structured cycles, the approach we propose is still able to return the predecessors and successors of an object, but it will not be applicable for the arbitrary cycles, because they will violate most of the well-formedness requirements discussed in Chapter 3. The functions introduced below are used in other rules introduced earlier in this chapter with some restrictions. For example, the functions `preds`, `imPreds`, `activePreds` used in rules `ANDFork` (in Section 3.4.2) and has similar structure of the concept but differs in the aim, i.e. `preds` retrieves the object's predecessor, `imPreds` retrieves the immediate predecessors for an object, `activePreds` retrieves a true if one (or more) of the predecessors is active, and `preds` which is discussed below retrieves the set of all predecessors starting from an object till the start event is reached.

Predecessors

In `preds` functions, a depth-first search mechanism is used in order to return all the predecessors of an object. It is defined to take the following arguments: an `Object`, which we want its predecessors, and the `ObjectSet` in which the object is a member. Moreover, the function takes two object sets as arguments with the value (`noobject`) in its initial state; first one to use as a temporary storage for objects with more than one predecessors (e.g. a merge gateway with more than one predecessor) till all the predecessors are fetched, and the other one is used to store the output objects till the procedure is finished. The function returns the set of predecessor objects if the `start` object is reached with no objects in the temporary storage object set, as defined below and as shown graphically in Figure C.1 (c).

```
op preds : Object ObjectSet ObjectSet ObjectSet -> ObjectSet .
eq preds(O, noobject, noobject, A) = A .
eq preds(< X : startEvent | AS1 >, A, noobject, B) = B .
```

The first equation above assumes that all the process objects have been retrieved (e.g. starting from the end event, and retrieving all the predecessors till the start event). In that case, the process object set will be empty (i.e. `noobject` operator) and the temporary storage is also empty, and the function returns the output set of objects (`A`) containing the set of all the predecessors. The second equation above assumes another scenario, where the start event is reached and the temporary storage is empty (i.e. `noobject` operator), hence, the function returns the set of predecessors.

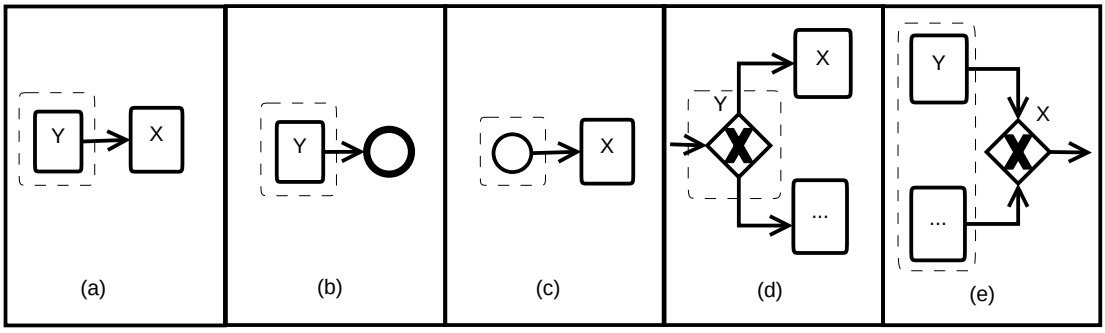


Figure C.1: BPMN illustration for different cases considered by the `preds` function

The `preds` function considers many cases for the precedence relationships in a business process model. The general forms of these cases are demonstrated using BPMN notation in Figure C.1 which will be used below to explain the function definition. The cases in (a) and (b) in Figure C.1 show the predecessor of an activity (or an end event) which is a single object. The function checks here that the input transition for the object `X` is the same as the output transition

of the object Y and this declared them as Y is the immediate predecessor of X. The equation below shows that object X is the query object, object Y is its predecessor, we know that because X's input transition is the same as Y's output transition, (i.e. t N1), and A is the rest of the ObjectSet. The equation proceeds by copying Y to the output object set C, and replacing X by Y as the query object. In this case X has only one predecessor (Y). Notice that the equation below rewrites X into Y as X has only one predecessor (i.e. Y) and it is Y's turn to bring its predecessors.

```
eq preds(< X : K | in:tN1;AS1 >,(< Y : L | out:tN1;AS2 >,A),B,C)
 = preds(< Y : L | out:tN1;AS2 >,A,B,(< Y : L | out:tN1;AS2 >,C)) .
```

The following equation definition supposes that the object X has more than one predecessors (e.g. a merge gateway like the one in Figure C.1 (e)) and Y is one of them. In this case, Y is being copied to the output object set C as part of the output, and to B as an object waiting to get its predecessors afterwards. Object X stays in the query object position after removing the input transition which fetch Y as a predecessor. As we still have X as a query object and removing one of its input transitions each step we fetch one of its predecessors, after some steps it will have no other transitions (notrans). In this case we need to remove X as a query object and put one of it predecessors stored with B. Note that we are not adding Y to the output object set C, because it is already there from a previous step. Also, the output transitions for Y is a set of transitions (tN1,T1) giving the possibility of Y being a splitting gateway with more than one output transitions. In the Figure C.1 we used the XOR gateway in cases (d) and (e) for illustrating the idea and the same concept applies to the AND and OR gateways.

```
eq preds(<X:K|in:(tN1,T1);AS1>,(<Y:L|out:tN1;AS2>,A),B,C)
 = preds(<X:K|in:T1;AS1>,A,(<Y:L|out:tN1;AS2>,B),
    (<Y:L|out:tN1;AS2>,C)) [owise] .
eq preds(<X:K|in:notrans;AS1>,A,(<Y:L|out:(tN1,T1);AS2>,B),C)
 = preds(<Y:L|out:(tN1,T1);AS2>,A,B,C) .
```

If the predecessor (Y in the equations below) has more than one output transitions (e.g. the XOR split gateway in Figure C.1 (d)), then it will be referenced in more than one object predecessor sets while it is already in the output set of predecessors. In the first check, Y is the only predecessor for X, so the X is removed from the query and Y has been rewritten to the query and added to the output set of predecessors C. This illustrated by the first equation below. This is followed by adding the split gateway into the temporary storage B to return all the objects that it is a predecessor for as shown in the second equation below. This done by removing one transition

at a time when its connected object is retrieved. Finally, the third equation below checks, with the last predecessor, if it is already in the output set or not, if it is in the output set, then it is not duplicated and the object in query is updated to the first of the retrieved predecessors Y.

```
eq preds(< X : K | in : t N1 ; AS1 >,
      (< Y : L | out : (t N1 , T1) ; AS2 >, A), B, C)
 = preds(< Y : L | out : (t N1 , T1) ; AS2 >, A, B,
(< Y : L | out : (t N1 ,  T1) ; AS2 >, C)) .
eq preds(< X : K | in : (t N1 , T1) ; AS1 >,
  (< Y : L | out : (t N1 , T2) ; AS2 >, A), B, C)
 = preds(< X : K | in : T1 ; AS1 >, A,
      (< Y : L | out : (t N1 , T2) ; AS2 >, B),
    (< Y : L | out : (t N1 , T2) ; AS2 >, C)) .
eq preds(< X : K | in : t N1 ; AS1 >, A,
(< Y : L | out : (t N1 , T1) ; AS2 >, B),
(< Y : L | out : (t N1 , T1) ; AS2 >, C))
 = preds(< Y : L | out : (t N1 , T1) ; AS2 >, A, B,
(< Y : L | out : (t N1 , T1) ; AS2 >, C)) [owise] .
```

If the start event is reached while the temporary storage has objects in it, then the start event is moved to the output object set and one of the objects in the temporary storage is moved to the query to bring its predecessors. Check the case (c) in Figure C.1 and the rule below which model this particular situation.

```
eq preds(< X : startEvent | AS1 >, A,
  (< Y : L | AS2 >,B),
  (< X : startEvent | AS1 >, < Y : L | AS2 >,C) )
 = preds(< Y : L | AS2 >, A, B,
  (< X : startEvent | AS1 >, < Y : L | AS2 >, C)) .
```

Successors

A similar approach, as the one used in defining the preds function above, is followed in defining the succs function. However, the succs function is mainly used to retrieve the set of all successors of an object in the business process till the end event. A depth-first approach is used to find the set of objects that comes after an object (its successors). The stopping condition of

the function is when it reaches the end event without remaining objects in its temporary storage object set taking into account that according to the well-formed BPMN definition in Section 3.3, a BPMN process has only one end event. The function is defined to take as arguments the object in query, the whole object set, the temporary storage object set, and the output storage object set.

```
op succs : Object ObjectSet ObjectSet ObjectSet -> ObjectSet .
eq succs(O, noobject, noobject, A) = A .
eq succs(< X : endEvent | AS1 >, A, noobject, B) = B .
```

a similar definition is given below for the function `succs`.

```
eq succs(< X : K | out : t N1 ; AS1 >,
      (< Y : L | in : t N1 ; AS2 >, A), B, C)
 = succs(< Y : L | in : t N1 ; AS2 >, A, B,
(< Y : L | in : t N1 ; AS2 >, C)) .
eq succs(< X : K | out : (t N1 , T1) ; AS1 >,
(< Y : L | in : t N1 ; AS2 >, A), B, C)
 = succs(< X : K | out : T1 ; AS1 >, A,
   (< Y : L | in : t N1 ; AS2 >, B),
    (< Y : L | in : t N1 ; AS2 >, C)) [owise] .

eq succs(< X : K | out : t N1 ; AS1 >,
(< Y : L | in : (t N1 , T1) ; AS2 >, A), B, C)
 = succs(< Y : L | in : (t N1 , T1) ; AS2 >, A, B,
(< Y : L | in : (t N1 , T1) ; AS2 >, C)) .
eq succs(< X : K | out : (t N1 , T1) ; AS1 >,
(< Y : L | in : (t N1 , T2) ; AS2 >, A), B, C)
 = succs(< X : K | out : T1 ; AS1 >, A,
   (< Y : L | in : (t N1 , T2) ; AS2 >, B),
(< Y : L | in : (t N1 , T2) ; AS2 >, C)) [owise] .

eq succs(< X : K | out : t N1 ; AS1 >, A, noobject,
(< Y : L | in : (t N1 , T1) ; AS2 >, C))
 =  < Y : L | in : (t N1 , T1) ; AS2 >, C .
eq succs(< X : K | out : (t N1 , T1) ; AS1 >, A, B,
```

```
(< Y : L | in : (t N1 , T2) ; AS2 >, C))
 = succs(< X : K | out : T1 ; AS1 >, A, B,
   (< Y : L | in : (t N1 , T2) ; AS2 >, C)) [owise] .

eq succs(< X : K | out : t N1 ; AS1 >, A,
(< Y : L | in : (t N1 , T1) ; AS2 >, B),
(< Y : L | in : (t N1 , T1) ; AS2 >, C))
 = succs(< Y : L | in : (t N1 , T1) ; AS2 >, A, B,
(< Y : L | in : (t N1 , T1) ; AS2 >, C)) .
eq succs(< X : K | out : notrans ; AS1 >, A,
(< Y : L | in : (t N1 , T1) ; AS2 >, B), C)
 = succs(< Y : L | in : (t N1 , T1) ; AS2 >, A, B, C) .
eq succs(< X : endEvent | eventType : end ; AS1 >, A,
(< Y : L | AS2 >, B),
(< X : endEvent | eventType : end ; AS1 >,< Y : L | AS2 >,C))
 = succs(< Y : L | AS2 >, A, B,
   (< X : endEvent | eventType : end ; AS1 >,< Y : L | AS2 >,C)) .
```

Identifying Responsibilities

In order to retrieve all the objects in a certain pool, or to answer a question of what are the activities that a certain participant is responsible for in a business process, we defined the function `OinP`. This function takes the name of the pool (participant) as a String and the set of all objects that included into the process under consideration. It retrieves the set of objects which do belong to that specific pool in query. The definition in Maude is below.

```
op OinP : String ObjectSet ObjectSet ->  ObjectSet .
eq OinP(S1, noobject, B) = B .
eq OinP(S1,(< X : K | pool : S1 ; AS1 >,A),B)
 = OinP(S1,A,(< X : K | pool : S1 ; AS1 >,B)) .
eq OinP(S1,(< X : K | pool : S2 ; AS1 >,A),B)
 = OinP(S1,A,B) [owise] .
```

The function `OinP`, defined above, takes the name of the pool (i.e. a participant in the process, which can be a person, a role, a department, or a company), and the whole process objects. It uses the second `ObjectSet` argument as a storage for the output objects. The first equation defines

the situation of having no objects in the process, and thus the function retrieves the set of objects in the output storage `B`. The second equation above, matches the `pool` value in an object in the process with the String value `S1` in query. If there is a match, then the object `X` is moved to the output storage and removed from the process. Finally, in the third equation, the otherwise situation is defined (`[owise]`). If an object in the process does not have the matching `pool` name, then it is removed from the process object set and will not be added to the output storage as well.

Function `into` is used to check the existence of an object into an object set and defined as follows.

```
op into : Object ObjectSet -> Bool .
eq into(O, (O,A)) = true .
eq into(O,A) = false [owise] .
```

C.2 WFS functions

The well-formedness check introduced in Section 3.3 was:

```
subsort WFprocess < ObjectSet .
op wfs : Object ObjectSet ~> Bool .
var O : Object .
var A : ObjectSet .
ceq wfs (O,A) = true
 if wfstartendTF((O,A),noobject) /\ wfExceptionTF((O,A),noobject) /\
 wfActivityTF((O,A),noobject) /\ wfGatesTF((O,A),noobject) /\
 wfpathTF(O,(O,A),noobject) .
eq wfs(O,A) = false [owise] .
```

In the second part of the condition, function `wfExceptionTF` is about the exception events; that an exception event should have no incoming transitions and only one outgoing transition. Functions `exceptionCollector`, `wfException` and `wfExceptionTF` are used to collect the exception events in the process, collect the well-formed exception events in the process and check the equality of the two sets of object respectively. The condition evaluates to *true* if the two sets have the same elements; hence all the exception events in the process satisfies the well-formed exception requirements.

```
op exceptionCollector : ObjectSet ObjectSet -> ObjectSet .
```

```
eq exceptionCollector((<E1:intermediateEvent|eventType:exception;
   AS1>,A),B)
 = exceptionCollector(A,(<E1:intermediateEvent|eventType:exception;
   AS1>,B)).
eq exceptionCollector(A,B) = B [owise] .

op wfException : ObjectSet ObjectSet -> ObjectSet .
eq wfException((<E1:intermediateEvent|eventType:exception;
 sourceObject:Y;in:notrans;out:tN1;AS1>,<Y:K|error:tN1;AS2>,A),B)
= wfException(A,(<E1:intermediateEvent|eventType:exception;
 sourceObject:Y;in:notrans;out:tN1;AS1>,B)) .
eq wfException(A,B) = B [owise] .

op wfExceptionTF : ObjectSet ObjectSet -> Bool .
ceq wfExceptionTF(A,noobject) = true
 if wfException(A,noobject) = exceptionCollector(A,noobject) .
eq wfExceptionTF(A,B) = false [owise] .
```

The third part of the condition contains the function `wfActivitiesTF` and covers the second requirement in Definition 3.3.2 about the number of incoming and outgoing transitions of an activity or non-exception event. The functions `ActivityCollector`, `wfActivity` and `wfActivityTF` are used to collect the activities and non-exception events in the process, collect the well-formed ones in the process and check the equality of the two sets of object respectively. The condition evaluates to *true* if the two sets have the same elements; hence all the activities and non-exception events in the process are well-formed.

```
op ActivityCollector : ObjectSet ObjectSet -> ObjectSet .
eq ActivityCollector((< X : task | AS1 >,A),B)
 = ActivityCollector(A,(B,< X : task | AS1 >)) .
eq ActivityCollector((< X : subprocess | AS1 >,A),B)
 = ActivityCollector(A,(B,< X : subprocess | AS1 >)) .
eq ActivityCollector((< X : intermediateEvent | eventType : message;
   AS1 >,A),B)
 = ActivityCollector(A,(B,< X : intermediateEvent | eventType :
   message;AS1>)) .
```

```
eq ActivityCollector(A,B) = B [owise] .

op wfActivity : ObjectSet ObjectSet -> ObjectSet .
--- a task
eq wfActivity((< X : task | in : tN1 ; out : tN2 ; AS1 >,A),B)
 = wfActivity(A,(< X : task | in : tN1 ; out : tN2 ; AS1 >,B)) .
--- a subporcess
eq wfActivity((< X : subprocess | in : tN1 ; out : tN2 ; AS1 >,A),B)
 = wfActivity(A,(< X : subprocess | in : tN1 ; out : tN2 ; AS1 >,B)) .
--- a message
eq wfActivity((< X : intermediateEvent | eventType:message ; in:tN1;
 out : tN2 ; sourceObject : Y ; targetObject : Z ; AS1 >,A),B)
 = wfActivity(A,(< X : intermediateEvent | eventType:message ; in:tN1;
  out : tN2 ; sourceObject : Y ; targetObject : Z ; AS1 >,B)) .
eq wfActivity(A,B) = B [owise] .

op wfActivityTF : ObjectSet ObjectSet -> Bool .
ceq wfActivityTF(A,noobject) = true
 if wfActivity(A,noobject) = ActivityCollector(A,noobject) .
eq wfActivityTF(A,B) = false [owise] .
```

For the gateways well-formedness, we use a similar approach, except it has different cases for different gateways. Function `gateCollector` collects all the gateways in the process. Function `wfGates` collects the well-formed gateways from the process. It has to fulfil the requirements for each gateway type. For example, the AND-fork gateway should have one incoming transition and more than one outgoing transitions to be considered well-formed. The following is the related `wfGates` function. It restricts the transfer of the gateway from the process set of objects to the well-formed set of gateways by having exactly one incoming transition `tN1`, and more than one outgoing transition `tN2,T1` where `T1` is a set of transitions that should not be empty (`T1 =/= notrans`).

```
op gateCollector : ObjectSet ObjectSet -> ObjectSet .
eq gateCollector((< G1 : G | AS1 >,A),B)
 = gateCollector(A,(< G1 : G | AS1 >,B)) .
eq gateCollector(A,B) = B [owise] .
```

```
op wfGates : ObjectSet ObjectSet -> ObjectSet .
ceq wfGates((< G1 : aforkgate | in:tN1;out:(tN2,T1);AS1 >,A),B)
= wfGates(A,(< G1 : aforkgate | in:tN1;out:(tN2,T1);AS1 >,B))
  if T1 =/= notrans .
ceq wfGates((< G1 : xsplitgate | in:tN1;out:(tN2,T1);AS1 >,A),B)
= wfGates(A,(< G1 : xsplitgate | in:tN1;out:(tN2,T1);AS1 >,B))
  if T1 =/= notrans .
ceq wfGates((< G1 : osplitgate | in:tN1;out:(tN2,T1);AS1 >,A),B)
= wfGates(A,(< G1 : osplitgate | in:tN1;out:(tN2,T1);AS1 >,B))
  if T1 =/= notrans .
```

XOR-split and OR-split gateways have the same condition for well-formedness (see third and fourth requirement in Definition 3.3.2). In case of join/merge gateways, the gateway should have exactly one outgoing transition and more than one incoming transitions.

```
ceq wfGates((< G1 : ajoingate | in:(tN2,T1);out:tN1;AS1 >,A),B)
  = wfGates(A,(< G1 : ajoingate | in:(tN2,T1);out:tN1;AS1 >,B))
  if T1 =/= notrans .
ceq wfGates((< G1 : xmergegate | in:(tN2,T1);out:tN1;AS1 >,A),B)
  = wfGates(A,(< G1 : xmergegate | in:(tN2,T1);out:tN1;AS1 >,B))
  if T1 =/= notrans .
ceq wfGates((< G1 : omergegate | in:(tN2,T1);out:tN1;AS1 >,A),B)
  = wfGates(A,(< G1 : omergegate | in:(tN2,T1);out:tN1;AS1 >,B))
  if T1 =/= notrans .
```

After that we apply the function `wfGatesTF`, which tests if the two resulted sets of objects (from functions `gateCollector` and `wfGates`) are the same (i.e. well-formed gateways) or not.

```
op wfGatesTF : ObjectSet ObjectSet -> Bool .
ceq wfGatesTF(A, noobject) = true
 if wfGates(A, noobject) == gateCollector(A, noobject) .
eq wfGatesTF(A, B) = false [owise] .
```

Finally, function `wfpathTF` in used in the last condition in the main well-formedness query. It retrieves *true* if the set of object's predecessors contains the start event object and the set of object's successors contains the end event object. The goal is to test all the objects in a process to

follow the requirement of being on a path from the start event to the end event. That means, if a process has a dangling edge (or activity) or does not have a start/end event, this will be discovered with at least one object in the process query. This will give the modeller an idea about where is the problem to fix.

```
op ObjCid : Object -> Cid .
eq ObjCid (< X : K | AS1 >) = K .
op wfpathTF : Object ObjectSet ObjectSet -> Bool .
ceq wfpathTF(O, (O1, A, O2), noobject) = true
 if (ObjCid (O1) == startEvent) /\
    O1 in (preds(O,(O1,A),noobject,noobject)) /\
    (ObjCid (O2) == endEvent) /\
    O2 in (succs(O,(O2,A),noobject,noobject)) .
eq wfpathTF(O, A, B) = false [owise] .
```

C.3 AND Rules Functions

The `activateANDsuccessors` function definition specifies that the function will search for the objects which has incoming flow matches the outgoing flow from the AND fork gateway. If a matching is found, then its object's states is changed from inactive to active if there are no execution conditions exist (i.e. `cond:false`). If there are execution conditions found (i.e. `cond:true`), the object status is changed from inactive to *ready2bActive* (i.e. `ToBeActive:true`). Otherwise state (`[owise]`) if the object does not form a match, then it is left with no change of its state. Finally, the function retrieves the updated object set `A` if there is no more transitions in the first argument, meaning no other immediate successors for the AND join gate to activate.

```
op activateANDsuccessors : TransSymbol ObjectSet -> ObjectSet .
eq activateANDsuccessors(notrans,A) = A .
eq activateANDsuccessors((tN1,T1),(< X : K | in:(tN1,T2); active:
   false; cond : false; ToBeActive : false; AS1 >,A))
 = activateANDsuccessors((T1),(< X : K | in:(tN1,T2); active:true;
   cond : false; ToBeActive : false; AS1 >,A)) .
eq activateANDsuccessors((tN1,T1),(< X : K | in:(tN1,T2); active:
   false; cond : true; ToBeActive : false; AS1 >,A))
 = activateANDsuccessors((T1),(< X : K | in:(tN1,T2); active:false;
```

```
    cond : true; ToBeActive : true; AS1 >, A)) [owise] .
```

In order to apply the `ANDjoin` rule in Section 3.4.2, the AND join object should be in a *ready to be active* state, which indicated using the attributes (`cond:true` and `ToBeActive :true`). The function `activePreds` used in the rule `ANDjoin` is used to assure that *all* the predecessors of an AND join gateway are active. It takes the set of input transitions `(t N1,T1)` for the AND join gateway and the whole object set. If the connected predecessor is `inactive` (or `ToBeActive`), the function retrieves *false*, otherwise, it retrieves *true*, as detailed below.

```
 op activePreds : TransSymbol ObjectSet -> Bool .
 eq activePreds(notrans,A) = true .
 eq activePreds((tN1,T1),(< X : K | out:(tN1,T2);active:false;
    ToBeActive : false ; AS1 >,A)) = false .
 eq activePreds((tN1,T1),(< X : K | out:(tN1,T2);active:false;
    ToBeActive : true ; AS1 >,A)) = false .
 eq activePreds((tN1,T1),(< X : K | out:tN1;active:true;AS1>,A))
= activePreds(T1,(< X : K | out:tN1;active:true;AS1>,A)) .
 eq activePreds(T1,A) = false [owise] .
```

C.4 Decision Gateways rules Functions

In rule `XORsplit`, we use the function `activateXORchild` to activate the XOR chosen child. The arguments are in order: the outgoing transitions of the XOR-split gateway (sort `TransSymbol`), the default transition defined in the XOR-split gateway (sort `TransSymbol`), the error transition defined in the XOR-split gateway (sort `TransSymbol`), the associated guard expressions (sort `Gexp`), the control values (sort `ControlValue`), and the set of all objects in the process (sort `ObjectSet`). It retrieves the set of objects in the process after activating the corresponding XOR-split child. The following conditional equation is used to activate the XOR-split gateway successor after evaluating the guard expressions. For example, in Figure C.2, if the flow containing the guard expression for the flow entering activity `a` is successfully evaluated (to *true*), then the activity `a` is activated next.

```
  op activateXORchild : TransSymbol TransSymbol TransSymbol
     Gexp ControlValue ObjectSet -> ObjectSet .
 ceq activateXORchild((t N1,T1),(T3),(T2),GExp,CVs,
```

```
      (<< < X : K | in : t N1; active : false; AS1 >, A >>))
   = << < X : K | in : t N1 ; active : true ; AS1 >, A >>
 if evalGuard (GExp,CVs,T3) = t N1 .
```

To evaluate the guard expression in an XOR gateway we use the function `evalGuard` which will be discussed later in Section 3.4.2. In the function `activateXORchild`, a *default flow* is activated if and only if none of the conditions evaluates to *true*. In case all conditions evaluates to false and a default flow has not been specified, an exception is thrown.

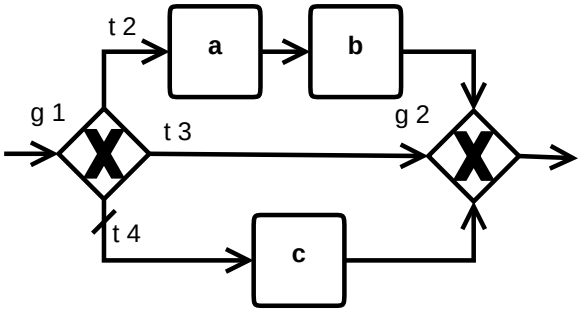


Figure C.2: Example model with XOR split and merge gateways

The first equation below defines the case of only one remaining transition (of the XOR split outgoing transitions) and it is the pre-defined default transition at the same time (activity `c` in Figure C.2). In this case the object attached to that transition is activated as the XOR-split successor (child).

```
eq activateXORchild((t N1),(t N1),(notrans),GExp,CVs,
 (<< < X : K | in : t N1 ; active : false ; AS1 >, A >>))
 = << < X : K | in : t N1 ; active : true ; AS1 > , A >> .
```

The following equation is also targeting the default flow, however, it is applied if no condition is evaluated to *true*, notice the `notrans` for the first function argument. In this case a default flow is the only choice. In Figure C.2, the default flow is the flow containing activity `c`, and it is supposed to have no conditional expressions.

```
eq activateXORchild(notrans,(t N1),(T1),GExp,CVs),
  (< X : K | in : t N1 ; active : false ; AS1 >, A))
 = << < X : K | in : t N1 ; active : true ; AS1 > , A >> .
```

Another form of the `actiavteXORchild` equation deals with a scenario of no condition is evaluated to *true* and no pre-defined default flow in the gateway at the same time. It leaves nothing

but throwing an exception event, which is attached to the gateway. Activating the exception may lead to unsuccessfully terminating the process, as there is no visible exception flow in the model. This case is not described in Figure C.2.

```
eq activateXORchild((notrans), (notrans), (t N1), (GExp),
   (CVs), (<< < E1 : intermediateEvent | eventType : exception ;
   in : notrans ; out : t N1 ; active : false ; AS1 >, A >>))
 = << < E1 : intermediateEvent | eventType : exception; in : notrans;
       out : t N1 ; active : true ; AS1 >, A >> .
```

The last form of the function is the *otherwise* form (`owise`) which completes the definition of the function to consider defined values for all possibilities. Therefore, if the function did not match any of the above forms, it will proceed to the next XOR successor as defined below.

```
eq activateXORchild((t N1,T1),(T2),(T3),GExp,CVs,A)
  = activateXORchild((T1),(T2),(T3),GExp,CVs,A) [owise] .
```

In rule `ORmerge`, the merge gateway predecessors should be deactivated in order to make the gateway active (i.e. completing the merge). The function sued for this is function `deactivateOPreds` defined below.

```
var Q : Bool .
op deactivateOPreds : TransSymbol ObjectSet -> ObjectSet .
eq deactivateOPreds(notrans, A) = A .
eq deactivateOPreds((t N1, T1),
   (< X : K | out : tN1 ; active : Q ; AS1 >, A))
 = deactivateOPreds((T1),
   (< X : K | out : t N1 ; active : false ; AS1 >,A)) [owise] .
```