IMT School for Advanced Studies, Lucca
Lucca, Italy

Exploiting Process Algebras and BPM Techniques for Guaranteeing Success of Distributed Activities

PhD Program in PHD in System Science

Track in Computer Science and Systems Engineering (CSSE)

XXXIII Cycle

By

Sara Belluccini

2023

The dissertation of Sara Belluccini is approved.

PhD Program Coordinator: ¡coordinator¿, IMT School for Advanced Studies Lucca

Advisor: Prof. Rocco De Nicola, IMT School For Advanced Studies of Lucca

Co-Advisor: Prof. Francesco Tiezzi, University of Florence

The dissertation of Sara Belluccini has been reviewed by:

Maurice H. ter Beek, ISTI-CNR, Pisa

Andrea Burattin, Danmarks Tekniske Universitet

IMT School for Advanced Studies Lucca
2023

Contents

List of Figures

List of Tables

Acknowledgements

Part of the work presented in this thesis has been previously published in two co-authored papers. In particular, Chapter 3 is based on [20], joint work with Rocco De Nicola, IMT School For Advanced Studies of Lucca, Marlon Dumas, University of Tartu, Pille Pullonen, Cybernetica AS, Barbara Re and Francesco Tiezzi, University of Camerino, in Proceedings of the 8th International Conference on Formal Methods in Software Engineering, IEEE/ACM, and on an extension of this paper that is currently under review. Chapter 4 is based on [18], a joint work with Rocco De Nicola, IMT School For Advanced Studies of Lucca, Barbara Re and Francesco Tiezzi, University of Camerino, for Integrated Formal Methods: 16th International Conference, Springer.

Abstract

The communications and collaborations among activities, processes, or systems, in general, are the base of complex systems defined as distributed systems. Given the increasing complexity of their structure, interactions, and functionalities, many research areas are interested in providing modelling techniques and verification capabilities to guarantee their correctness and satisfaction of properties. In particular, the formal methods community provides robust verification techniques to prove system properties. However, most approaches rely on manually designed formal models, making the analysis process challenging because it requires an expert in the field. On the other hand, the BPM community provides a widely used graphical notation (i.e., BPMN) to design internal behaviour and interactions of complex distributed systems that can be enhanced with additional features (e.g., privacy technologies). Furthermore, BPM uses process mining techniques to automatically discover these models from events observation. However, verifying properties and expected behaviour, especially in collaborations, still needs a solid methodology.

This thesis aims at exploiting the features of the formal methods and BPM communities to provide approaches that enable formal verification over distributed systems. In this context, we propose two approaches. The modelling-based approach starts from BPMN models and produces process algebra specifications to enable formal verification of system properties, including privacy-related ones. The process mining-based approach starts from logs observations to automati-

cally generate process algebra specifications to enable verification capabilities.

Chapter 1

Introduction

A distributed system consists of a collection of distinct processes that communicate by exchanging messages [74]. A process in a distributed system can be carried out by an organisation, a device, a single component, a person, or any element of which we consider its single behaviour to reason over its interaction with other similar processes for performing distributed activities. Examples of distributed systems can be found in many different application domains: wireless sensor networks measuring quality parameters of gases [81], distributed ledger technologies (e.g., the Bitcoin blockchain) supporting cryptocurrency transactions [120], home systems to perform homecare monitoring through the remote connection to the assisted people and their environment [66], and many more.

Already in 1985, in the Journal of "Communications of the ACM", Kleinrock wrote:

> *"Growth of distributed systems has attained unstoppable momentum. If we better understood how to think about, analyze, and design distributed systems, we could direct their implementation with more confidence."* [70]

After many years, this statement is not only still valid but it is even more relevant due to the complexity of today's distributed systems, such as those used in Industry 4.0 [68, 94], healthcare [72, 121], energy [126, 33], marine sector [40, 80], business [135, 9], and so on. The behaviour of a distributed system is not represented just by the activities of a single organisation or system acting alone but introduces cooperation and collaboration among more of them. This reveals the necessity of features like compatibility of distributed behaviour or correctness of the privacy measures put in place. Consequently, it is essential to design any distributed system properly, verify its overall behaviour and check that it maintains the expected behavioural and security properties [137].

Given the task's relevance, this thesis aims at answering the following research question:

> How can practitioners formally guarantee the success of distributed activities?

The Formal Methods (FM) research community has long been dedicated to studying and advancing the task of formal verification.

FM is, in fact, defined as "mathematics-based techniques for the specification, development, and (manual or automated) verification of software and hardware systems" [52] The specification phase provides a solid mathematical foundation to reason about systems by relying on process algebras (e.g., CCS [87], CSP [61], ACP [21]), formal languages (e.g., $mCRL2$ [55], Petri Nets [130]), and mathematical models (e.g., Labelled Transition Systems [42], Finite State Automata [17], Büchi Automaton [117]) to describe systems as a combination of subsystems and represent their interactions. In addition, verifying properties like deadlock, liveness or safety and comparing systems' behaviour is based on rigorous model checking techniques [17] and equivalence checking notions [92].

The modelling activities based on the above formalisms require expert modellers to design the systems' behaviour properly. Similarly, the property specification and their verification typically demand expertise in logical formalisms. This makes it hard for many practitioners from business and industrial context to use those techniques.

These limitations at a high level of abstraction highlight, firstly, the importance of having a system model that is both readable and easily understandable from a practitioner's perspective to support the formal verification during the design phase of the distributed activities. Secondly, they highlight the importance of automatically generating the formal specification of a system and its properties based on its behaviour when it is already up and running. This requires extracting the system behaviour from observations or event logs, which can be utilized to generate formal specifications for verification purposes. In this way, the model can capture the dynamic aspects of the system and derive formal specifications that accurately represent the system's properties.

The high-level modelling of distributed activities is an area of interest for multiple research communities, particularly the Software Engineering (SE) and Business Process Management (BPM) research communities. In the field of SE, various approaches have been developed, with the most notable ones being defined by the OMG Systems Modeling Language consortium [1]. The Unified Modeling Language (UML)[2] is a widely recognized visual modelling language designed for defining, envisioning, constructing, and documenting object-based virtual systems and devices [124]. UML provides different types of diagrams that can be used to describe various functionalities or states of the system being modelled. Another approach within software engineering is the Systems Modeling Language (SysML)[3], which is an extension of UML also defined by the OMG. SysML is considered a general-purpose graphical modelling language for specifying, analyzing, designing, and verifying complex systems that may encompass hardware, software, information, personnel, procedures, and facilities [96]. While UML is primarily focused on software solutions, SysML has been designed to model a broader range of systems [85] by incorporating additional diagrams and modifying existing ones.

On the other hand, the BPM community deals with distributed sys-

[1] http://www.omg.org/
[2] http://www.omg.org/spec/UML/
[3] http://www.omgsysml.org/

tems focusing on the processes, the so-called *business processes* [79], that regulate the activities undertaken by the involved participants. More specifically, distributed systems are modelled as collaboration diagrams, which are graphical models that can represent the internal behaviour of different entities, and their interactions [46] through the usage of BPMN (Business Process Management Notation) [95]. BPMN is an intuitive standard that has acquired a clear predominance among the notations proposed to model collaborative business processes.

Extensive research has been conducted by the communities to compare and evaluate various modelling approaches in order to understand their distinct characteristics and suitability for different purposes. In [93], the authors conducted a comparative study of UML, BPMN, and EPC (Event-driven Process Chain) [67]. The latter is another modelling language defined by the BPM community for specifying temporal and logical relationships between activities in a business process [85]. The objective of this comparison was to assess the readability, understandability, and usability of these languages. While EPC was recognized for its effective use of colours, and UML, particularly through its activity diagram, was praised for its simplicity, BPMN emerged as the most comprehensible language according to the majority of survey participants. This was primarily attributed to BPMN's representation capabilities, such as pools and lanes, which enable the clear delineation of roles in a process [93]. In [110], the authors aimed to model Multi-Robot Systems (MRS) using both SysML and BPMN. SysML was selected for its requirement diagram, which facilitates the definition of performance criteria, as well as the block definition and internal block diagram, which describe the main components of the system architecture. On the other hand, BPMN was chosen to represent the detailed logic of the MRS system, as it extends the notations, semantics, and syntax of SysML and UML activity diagrams [110]. In [90], the authors compared BPMN to OWL (Web Ontology Language) and SysML, describing BPMN as being closer to a simulation language compared to OWL, and easier to learn in comparison to SysML. They emphasised the practical usefulness of BPMN as a reasoning tool for modellers, aiding them in mastering the complexity of real systems [90].

Considering the ease of use, understandability, and powerful representation capabilities of BPMN, it has been selected as the high-level modelling language for this thesis, serving as a valuable tool to address the research question at hand. Moreover, distributed systems, in general, and collaborative business processes, in particular, are often subject to stringent security and privacy requirements [23, 104]. For example, in a collaborative business process, each organisation should consider how it handles private data internally and how it exchanges private data or part of it with other organisations. To address these challenges, an extension of the standard notation, named PE-BPMN (Privacy-Enhancing BPMN) [104, 105], has been introduced.

However, while PE-BPMN allows analysts to include the specification of privacy mechanisms in their models, techniques are still missing to verify that the resulting privacy-enhanced process models accomplish the intended privacy properties and that their design is correct. For example, methods and tools are missing for detecting privacy leakages, i.e., states where a peer in the collaborative process may unduly gain access to private information.

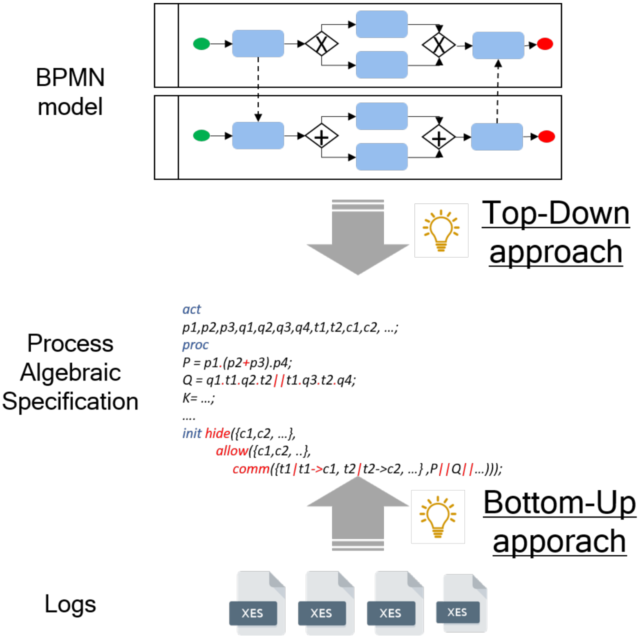
In the field of automated generation of formal specifications, the FM community has primarily focused on applying automata learning techniques [4, 64, 10, 91, 134, 60, 59]. These techniques involve constructing an automaton by providing inputs (such as traces) to a system and observing the corresponding outputs. On the other hand, the SE community has focused on generating finite state machines or graph models to support program comprehension and test case generation [22, 51, 73, 118]. In this context, the SE community attempts to infer models by analysing event logs. Although these techniques aim to synthetically reproduce a model from observations of system behaviour, the BPM community, particularly through process mining techniques, has demonstrated an active involvement in developing not only algorithms but also effective tools for use in business environments [11, 32, 127, 114]. This indicates a strong and practical application of process mining in real-world scenarios.

Process mining techniques are used to automatically discover process models from event data, shifting the focus from a process modelling to a process-centric data-driven approach [130]. The goal of process mining, indeed, is to extract process-related information by observing events recorded by some enterprise system [130]. However, even if this kind of technique allows to free the system modellers from the burden of explicitly modelling the behaviour of an existing system, there is still a lack of methods to ensure system properties, especially when considering issues specific to the collaboration level, i.e., those problems due to erroneous or unexpected interactions among the system components.

Based on the chosen approaches, *the* ***objective*** *of this thesis is to investigate the effective integration of techniques from the FM and BPM domains to enable robust verification capabilities for ensuring the successful execution of distributed activities.*

1.1 Research Objectives

The integration of FM and BPM aims to enhance the capabilities of verifying the correctness of distributed activities by combining powerful tools from both research communities. As highlighted above, the FM community is renowned for providing robust verification techniques to analyse complex behaviour and properties of distributed systems. However, the design phase of such systems still heavily relies on expert knowledge, which restricts the wider adoption of these approaches. On the other hand, BPM, with its widely adopted graphical notation and support, facilitates the design of distributed activities within the business environment. Additionally, process mining techniques enable the automatic discovery of process models by observing the behaviour of these activities. Nevertheless, verifying the expected behaviour, particularly in the context of collaborations between multiple entities, requires a solid methodology. By integrating FM and BPM, the aim is to leverage the strengths of each community to address the limitations and challenges associated with verifying distributed activities. The goal is to combine



```
act
p1,p2,p3,q1,q2,q3,q4,t1,t2,c1,c2, ...;
proc
P = p1.(p2+p3).p4;
Q = q1.t1.q2.t2||t1.q3.t2.q4;
K= ...;
....
init hide({c1,c2, ...},
       allow({c1,c2, ..},
          comm({t1|t1->c1, t2|t2->c2, ...} ,P||Q||...)));
```

Figure 1: Top-down and bottom-up approaches.

the robust verification techniques of FM with the powerful design and discovery capabilities of BPM, ultimately providing a comprehensive framework for ensuring the correctness and success of distributed activities in different phase.

To achieve this objective, we applied the following two approaches (see Figure 1):

- **Top-down approach**. Starting from models of distributed systems created using the BPMN standard notation, we automatically generate formal specifications that enable formal property verification.

- **Bottom-up approach**. Starting from logs of distributed systems' executions, we automatically generate formal specifications of systems' behaviour to enable their analysis.

In the context of combining different research communities to enhance the effectiveness of verifying distributed activities, a comprehensive investigation of both top-down and bottom-up approaches was deemed necessary. These approaches offer distinct advantages and find relevance in various scenarios.

The top-down approach proves well-suited for capturing the overall system design and comprehending high-level requirements, facilitating early identification of critical aspects. For example, this approach has been applied in the healthcare domain to model the patient care delivery process, specifically mapping the critical patient pathway to identify bottlenecks and potential causes of increased lead times and homecomings [8]. A similar application has been seen in the security analysis of cyber-physical systems, where a threat model that specifies MITM attacks is formalised to drive a CPS into an undesired state [75]. The top-down approach proves advantageous in supporting the design and verification phases of processes or systems that are not yet implemented or lack available data due to limitations in data collection or inherent difficulty.

That means if the system to be analysed is brand new, the top-down approach is more suitable because it can be applied without waiting for the first prototypical implementation of the system to be released. In other words, the approach can already be applied during the design phase of the new system. It allows verification of the correctness of the designed models in terms of expected behaviour. For example, a correct booking system design should ensure that the system will not send a confirmation email if the added payment method is not accepted. In addition, considering privacy concerns, a correct system design using the secret sharing technology should guarantee that no violation of the secret is expected at design time.

On the other hand, the model-driven top-down approach is unsuitable when the system is all up and running. In this scenario, an event data-based, bottom-up approach allows the engineer to understand and analyse what the system is doing, identifying possible unexpected behaviour. The bottom-up approach can effectively test and validate pre-existing components. By focusing on individual components and their interactions, this approach mitigates the risk of non-compliance with modelled requirements during execution of the actual implementation. Possible applications can vary from the healthcare domain to check whether clinical guidelines and protocols have been followed by inspecting real event logs describing it, manufacturing field, to discover hidden relationships in a production process given its event logs or to analyse maintenance processes and activities, education domain and so on [113]. The ability to derive models from running processes allows to validate the design's effectiveness, identify unexpected or inefficient behaviour and verify expected properties on the system or process under test.

The two approaches will be applied in the following running examples to show the effectiveness of their capabilities in verifying the correctness of distributed activities.

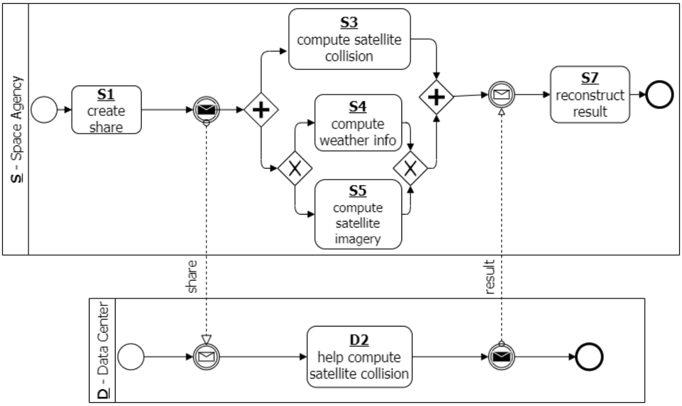


Figure 2: Running example of a distributed activity represented using the BPMN standard

The example in Figure 2, represented using a BPMN collaboration diagram, describes the distributed activity of a Space Agency using a service made available by a Data Centre to compute information about a satellite collision, and a Data Centre, providing its computing power to help in the computation.

Following, the control flow of the example is informally presented (more details about BPMN elements are given in Section 2.2). The Space

Agency splits the satellite data into two separate shares ($S1$) and then sends one of them to the Data Centre via a message called $share$. As soon as the Space Agency sends the message, it computes a satellite collision ($S3$) in parallel with the computation of either the weather info ($S4$) or the satellite image ($S5$). When the Data Centre receives the data, it executes some computation related to the collision ($D2$) and then sends the result back to the Space Agency via a message called $result$. In the end, the Space Agency receives and reconstructs ($S7$) the result. The following Chapters will present the results of the top-down and bottom-up approaches applied to the running example.

1.2 Contribution

In this thesis, we provide an instantiation of the two-approach methodology previously discussed by resorting to an effective combination of FM and BPM techniques.

Our instantiation of the *bottom-up approach* is inspired by process mining techniques to automatically generate a formal specification, written in $mCRL2$ language, from logs.

Our *top-down approach* instantiation is a model-driven technique that translates BPMN models into $mCRL2$ specifications. In addition, our model-driven technique also deals with PE-BPMN models, i.e. BPMN models enriched with the specification of privacy-enhancing technologies. This allows to verify that the mechanisms used to manage privacy have been used correctly at the design level. Notably, this is an important novelty in considering PETs within the BPMN context; indeed, while several formalisations of BPMN exist, there were no formalisation tailored explicitly to PE-BPMN.

The common point between the two approaches is the use of the $mCRL2$ language for specifying the distributed system's behaviour at the formal level. This enables reasoning on the properties of the system's behaviour, regardless of how it has been obtained, using the same set of techniques and tools, taking advantage of the rich and powerful toolset of $mCRL2$ for verifying system properties.

1.3 Thesis Structure

The rest of the thesis is structured into five chapters as follows:

- **Chapter 2 - Background Notions**: it describes all the background notions necessary to understand the scope and contribution of the thesis. Specifically, the chapter illustrates: the $mCRL2$ formal specification language used for the output of our top-down and bottom-up approaches; the BPMN notation and its extension PE-BPMN, used as starting point of the top-down approach; and process mining techniques, from which we took inspiration to develop our bottom-up approach.
- **Chapter 3 - From Collaboration Models to Formal Specifications**: it presents the top-down approach used to verify (PE-)BPMN models by translating them into $mCRL2$ specifications.
- **Chapter 4 - From Collaboration Logs to Formal Specifications**: it describes the bottom-up approach used to verify the behaviour of existing systems by generating a $mCRL2$ specification from their logs.
- **Chapter 5 - Related Work**: it contextualises our research contribution with respect to the related work about formal verification of BPMN and PE-BPMN models and the automatic generation of formal specifications from system observations.
- **Chapter 6 - Conclusions and Future Work**: it summarises the thesis contributions and presents future possibilities to further integrate formal methods and business process management solutions towards the successful verification of distributed activities.

Chapter 2

Background Notions

In this chapter, we will introduce all the background notions needed to understand the research question and the approaches used to answer it.

A formal specification language is a tool to model and verify the behaviour of complex systems that, thanks to its intrinsic compositional nature, is particularly effective on distributed systems. Moreover, allowing property verification enables its usage in various fields, such as health, business or natural environments.

First, we present basic notions of the formal language used to generate the models, that is $mCRL2$ [55, 27], a powerful language equipped with a toolset for modelling, validation and verification of systems.

Among the process algebra options available, $mCRL2$ was chosen primarily for its robust capability to handle data in the formalisation process. M provides, in addition to data handling features, available also in languages like PROMELA [62], the input language of the SPIN tool, the ability to declare specific types for data objects and use advanced data structures such as Sets or Bags. Another key advantage of $mCRL2$ is its rich toolset, which includes model-checking functionalities and supplementary features such as equivalence checking, LTS or trace visualisation, and other valuable features that enhance the verification phase not supported by similar to tools like SPIN [133] or UPPAAL [128]. Furthermore, the formal specification language and its associated toolset are actively maintained, and the community supporting its development remains highly active. Another reason for selecting $mCRL2$ is its support for time, which holds the potential for future integration into top-down and bottom-up generated models. Properties or formulas in $mCRL2$ are expressed using the mu-calculus [83].

Since our work aims to combine techniques from the FM and the BPM community, we will introduce the widely used notation for modelling BPM processes, namely BPMN [95], which is helpful to enable our model-driven approach, and the process mining discipline, its procedures and the main features from which the bottom-up approach is inspired.

2.1 $mCRL2$ Formal Specification Language

The *Micro Common Representation Language* ($mCRL2$) is the successor of μCRL, a specification language that extends ACP, adding notions like data, time, and multi-actions [57, 55]. $mCRL2$ is a powerful language that can be used to model and automatically analyse the behaviour of distributed systems. Its strength is given by the combination of the language with a toolset of verification methods that is still actively maintained (the last update was released in June 2022) [39]. $mCRL2$ is the link between the two approaches enabling verification over models and observed behaviour.

$mCRL2$ can be divided into data and process specification. Components of a distributed system often exchange messages containing data items among themselves and with the environment. We need a language to describe data types and their elements to be used in behaviour.

In $mCRL2$, data specification is defined by the usage of keywords:

- $sort$ are non-empty, possibly infinite sets used to define a new data type. Data types that are commonly used have been predefined, like Booleans, natural numbers, sets or lists

- map lists auxiliary functions over the defined and predefined sorts

- var express all the variables (name and type) used in equations

- eqn defines how the functions should operate

Listing 2.1 reports a data specification example defining a new data type, called O, that contains an arbitrary number of elements of type Nat, i.e. natural numbers. We specified that the union of two elements of type O produces a data set of type O.

```
sort O = Nat;
map
union : O # O -> Set(O);
var
o1,o2: O;
eqn
union(o1,o2) = {o1} + {o2};
```

Listing 2.1: Example of data specification defining the data type O.

The process specification defines the behaviour of the system being modelled. Since our work uses a fragment of the $mCRL2$ specification language, we will present just the needed syntax. The complete syntax can be found in [55].

Definition 2.1.1 ($mCRL2$ Process Specification Syntax)**.** *A process specification in $mCRL2$ is a pair $\langle q, E\rangle$ where E is a set of process equations of the form $Q = q$ where q is a process expression defined by the following grammar.*

$$
\begin{array}{lll}
q ::= & & \textit{(Process expression)} \\
| & \alpha & \textit{(action)} \\
| & \cdot_{j \in J} Q_j & \textit{(Sequence operator)} \\
| & +_{j \in J} Q_j & \textit{(Choice operator)} \\
| & ||_{j \in J} Q_j & \textit{(Parallel Composition operator)} \\
| & sum\ q_1, ..., q_n : sortName\,.\,Q & \textit{(Sum operator)} \\
| & c \rightarrow q_1 <> q_2 & \textit{(Conditional operator)} \\
| & \nabla(V, q) & \textit{(Allow operator)} \\
| & \Gamma(C, q) & \textit{(Communication operator)} \\
| & \tau(I, q) & \textit{(Hiding operator)} \\
| & Q & \textit{(Process equation call)}
\end{array}
$$

where α denotes an **action** either of the form a, with no parameter (including the silent action tau), or of the form $a(d_1, \ldots, d_n)$, with data expression parameters d_i; $sortName$ identifies a **sort**, which can be predefined or defined in a data specification; c is a condition over data parameters that returns $true$ or $false$ as result. V and I denotes sets of actions; C denotes a set of **communication expressions**, each one defining the renaming of multi-actions (i.e. communicating actions that occur simultaneously) to a single action and Q denotes a process identifier.

The process syntax denotes with $._{j \in J} Q_j$ the **sequence** of processes, with $+_{j \in J} Q_j$ the **choice** among processes, and with $||_{j \in J} Q_j$ the **interleaving** among processes. The **sum** operator $sum\ q_1, \ldots, q_n$: *sortName* . Q is a generalisation of the choice operator that permits to express in a concise way the choice between a (possibly infinite) number of processes, by instantiating in Q the placeholders $q_1, \ldots, q_n$ with values of type *sortName* (e.g., $sum\ n\ :\ Nat\, .\, a(n)$ is equivalent to the process $a(0) + a(1) + a(2) + \ldots$). The **conditional operator** $c \rightarrow q_1 <> q_2$ represents the execution of a process based on a data condition (e.g. $(semaphore = red) \rightarrow stop <> go$); the else part can be omitted. The **allow** operator $\nabla(V, q)$ (written as $allow$ in the machine-readable input language of the $mCRL2$ tool) defines the set of actions V that the process Q can execute; all the other actions, except for tau, are blocked. The **communication** operator $\Gamma(C, P)$ (written as $comm$ in the $mCRL2$ toolset) permits synchronising actions in Q according to the communication expressions C; for example, $comm(\{a|b \text{ -> } c\}, (a \,||\, b))$ says that the parallel actions a and b must communicate, resulting in a c action. The **hide** operator $\tau(I, q)$ (written as $hide$ in the input language of the $mCRL2$ tool) hides those actions produced by Q that are in I, i.e. it turns these actions into tau actions. Finally, Q permits to **call** a process definition of the form $Q = q$, where Q is a unique process identifier.

In the $mCRL2$ tool, process specification is defined by the usage of the following keywords:

- act contains the declaration of the actions used in the specifications,
- $proc$ lists the process declarations,
- $init$ is necessary for every $mCRL2$ specification, and it states which process the specification is actually representing.

Listing 2.2 contains an example of a vending machine process specifications where inserting a coin makes it possible to select coffee or cappuccino and, after that, the quantity of sugar (from 0 to 4).

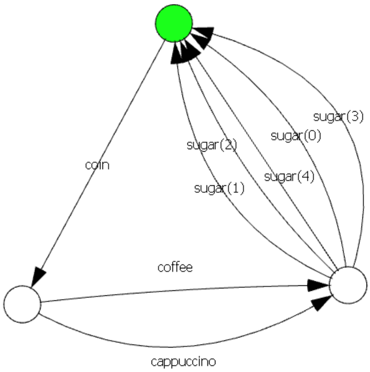


Figure 3: LTS of Listing 2.2

```
act
coin, coffee, cappuccino; sugar:Nat;
proc
VM = coin.(coffee + cappuccino).S.VM;
S = sum n: Nat.(n<5)->sugar(n);
init VM;
```

Listing 2.2: Example of process specification

The semantics of a process specifications in $mCRL2$ is defined through SOS rules (Structural Operational Semantics) represented as *labelled transition system (LTS)*, which consists of a set of states, a set of labels and transition relations representing the evolution steps of the specification. The SOS rules related to the above syntax can be found in Appendix A.

The $mCRL2$ specification language is supported by a toolset that provides equivalence and model checking functionalities.

2.1.1 Equivalence and Model Checking

Equivalence checking consists in comparing the behaviour of processes. For example, it can be used to check if the implementation of a system is consistent with respect to its abstract specification. Most of the equivalences are defined for LTS models.

$mCRL2$ supports different types of equivalences [43]: strong bisimilarity, weak bisimilarity, trace equivalence, weak trace equivalence, branching bisimilarity, strong simulation or divergence preserving branching bisimilarity.

The strongest relationship between models is *bisimilarity* and states that systems are equivalent if they can simulate each other at each step. Similarly, *weak bisimilarity* allows the execution of some silent action (tau) while simulating each other. For example, imagine a new LTS equal to the one in Figure 3 but with a new arrow executing the tau action after $coin$ and before the branch $coffee$ and $cappuccino$. Then the two LTSs are related by a weak bisimilarity equivalence. *Trace equivalence* indicates that two models can perform identical finite transition (or traces) sequences. As for weak bisimilarity, *weak trace equivalence* allows silent actions while still satisfying the relation. The definition of all the equivalences can be found in [57].

These notions will be used in Chapter 4 to compare the models generated from the logs using the bottom-up approach with the ones generated by commonly-used process mining approaches.

Model checking instead focus on verifying the properties of the system. The properties to be checked are specified in a first-order modal μ-calculus logic extended with data-dependent formulas. The μ-calculus logic is composed by *action formulas*, *regular formulas* and *state formulas* [83]. Following, we present part of its syntax that will be used to express properties over the generated formal specifications (the complete grammar can be found in [57]).

A **state formula** uses as underlining modal logic to express verification properties the Hennessy-Miner logic extended with fixed point modalities [57]:

$$\phi ::= true \mid false \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle a \rangle \phi \mid [a]\phi \mid \mu X.\phi \mid \nu X.\phi \mid X$$

The formula $true$ is true for each process state, and $false$ is never true. $\neg\phi$ (written as ! in the $mCRL2$ toolset) is the negation of ϕ, $\phi_1 \wedge \phi_2$ (written as && in the $mCRL2$ toolset) is valid when both ϕ_1 and ϕ_2 hold, and $\phi_1 \vee \phi_2$ (|| in the $mCRL2$ toolset) is valid when at least one formula hold. $\langle a \rangle \phi$ holds whenever it is true that if an action a can be performed, immediately after ϕ is valid. For example, taking into account the LTS in Figure 3 the property $\langle coin \rangle \langle coffee \rangle true$ means that the process VM can do the action $coin$ followed by an action $coffee$.

$\mu X.\phi$ is the minimal fixed point (mu in the $mCRL2$ toolset) that is valid for all those states in the smallest set X that satisfies the equation $X = \phi$; this is useful to define liveness properties stating that something good will happen within a finite number of steps. Finally, $\nu X.\phi$ (nu in the $mCRL2$ toolset) is valid for the states in the largest set X that satisfies $X = \phi$, which is helpful to define safety properties stating that something bad will never happen.

The μ-calculus uses regular formulas to express more than one action inside a single modality, which means that $\langle a \rangle \phi$ and $[a]\phi$ is actually $\langle \alpha \rangle \phi$ and $[\alpha]\phi$. To understand regular formulas, we start by defining action formulas. An **action formula** defines a set of actions:

$$\alpha ::= true \mid false \mid \forall\ d : Data.\alpha \mid \exists\ d : Data.\alpha \mid \bar{\alpha} \mid \alpha \cap \alpha \mid \alpha \cup \alpha$$

where $true$ and $false$ represent respectively the set of all actions and the empty set, which means that $\langle true \rangle \langle coffee \rangle true$ express that any action followed by an action coffee can be performed while $[true]false$ that no action can be performed. $\cap$ (&& in the $mCRL2$ toolset) denotes set intersection and $\cup$ (|| in the $mCRL2$ toolset) denotes set union of actions, i.e. $\langle coin \rangle \langle coffee || cappuccino \rangle true$ means that after the action $coin$ an action between $coffee$ or $cappuccino$ can be performed. $\bar{\alpha}$ (! in the $mCRL2$ toolset) denotes the complement of set action α. Finally, $\forall$ ($forall$ in the $mCRL2$ toolset) and $\exists$ ($exists$ in the $mCRL2$ toolset) are, respectively, the universal and existential quantifiers meaning that the formula holds if α holds for all values from the domain Data substituted for d in α, or α only needs to hold for some value in Data substituted for d.

Regular formulas extend the action formulas to allow the use of sequences of actions in modalities [57], their syntax is as follows:

$$R ::= nil \mid \alpha \mid R.R \mid R + R \mid R^* \mid R^+$$

where nil represents an empty sequence of actions, α is an action formula, $R.R$ denotes the concatenation of actions meaning that $\langle coin.coffee\rangle true$ it is equal to $\langle coin\rangle\langle coffee\rangle true$, while $R + R$ denotes the union of sequence of actions, i.e. $[coffee.coin + cappuccino.coin]false$ means that none of the sequences can happen. R^* is the transitive and reflexive closure of R and denotes that zero or more repetitions of the sequences in R can be performed. For example, $true^*$ means that any action, also none, can happen. Similarly, R^+ represents the transitive closure denoting one or more repetitions of the sequence R.

2.2 Business Process Model and Notation

The OMG standard **BPMN** [95] is currently acquiring a clear predominance among the notations proposed to model business processes. The BPMN elements presented in Figure 4 are the one supported by our top-down approach. They refer to BPMN collaboration diagrams, which focus on both intra- and inter-organisational aspects of business processes.

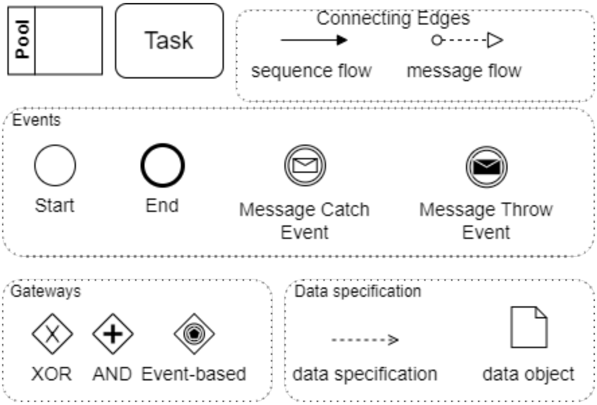


Figure 4: Elements of the BPMN Notation.

A *BPMN collaboration diagram* consists of a set of processes, each performed by an independent *party*. They are executed in parallel and synchronise via message exchanges (dashed arcs). Each process in a BPMN collaboration is enclosed in a pool (denoted as a rectangle). A process consists of tasks (rounded rectangles), events (circles) and gateways (diamonds), connected via sequence flows (directed arcs).

A *task* represents a logical unit of work. An *event* represents something triggered by the environment (e.g., start or end of process, incoming/outgoing message). Each task may be associated (via directed dotted arcs) with one or more input or output *data objects*. The intended meaning is that it reads the current state of each input object at its execution, and when it completes it writes into the output data objects.

A *gateway* is used to capture a non-deterministic choice (XOR gateway, marked by a "×"), or a deterministic choice (event-based gateway, marked by a pentagon marker inside a double line circle), or a parallel execution/synchronisation of multiple branches (AND gateway, marked with a "+"). A *sequence flow* indicates that the source element must be executed before the target element.

Considering collaborative business processes, like the running example in Figure 2, each organisation needs to consider how it handles private data internally and exchanges it with other organisations. Privacy requirements are critical in various types of processes, for example in healthcare processes where patient records are exchanged between peers, or in financial processes, where sensitive financial data moves across organisations with different access rights. For example, in Figure 2 the Space Agency could want to keep the data secret from the Data Centre that is computing it.

To address the challenges of designing collaborative systems that meet strict privacy requirements, existing modelling languages for collaborative systems need to be extended to capture privacy aspects [41]. Accordingly, several studies have proposed extensions of BPMN, to capture private data exchanges [108, 112]. In this work, we consider one such extension, namely PE-BPMN [104, 105].

2.2.1 PE-BPMN

PE-BPMN [104, 105] is a conservative extension of BPMN that allows designers to annotate tasks with stereotypes corresponding to different types of *privacy-enhancing technologies* (PETs).

PE-BPMN notation allows the designing of various privacy-enhancing technologies, such as secure communication or differential privacy. The privacy techniques supported by PE-BPMN are based on the classification of technologies in [41] that focus on the privacy and data protection goals that the use of the technologies has. Following, we present the subset of PETs consider in our work: *secret sharing*, *encryption* and *multi-party computation*

Secret sharing. This PET splits a secret into several shares that will be then distributed to a set of participants. The secret can be reconstructed by a participant if it has a sufficient number of shares [104]. To integrate secret sharing in a BPMN model, a task can be enanched with *SSSharing*, *SSComputation*, and *SSReconstruction* stereotypes, and related attributes, as reported in Figure 5.

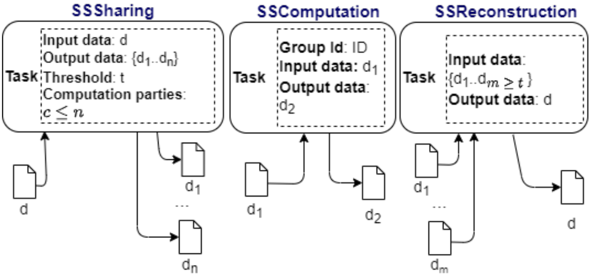


Figure 5: Secret sharing.

An *SSSharing* task takes in input a single data object, and generates the shares as data objects in output. This kind of task gives the possibility to define how many shares are needed to reconstruct the secret via the threshold attribute, and makes explicit how many computation parties will participate in the computation. Notice that the threshold has to be less than or equal to the number of computation parties, which has to be less than or equal to the number of outputs.

SSComputation indicates a task executing a computation over the share: it takes in input a share as a data object, and produces a share stored in a data object in output after the computation execution. For this stereotyped task we can set the group id attribute, which identifies all the tasks running the same computation over different shares.

Finally, *SSReconstruction* indicates a task able to reconstruct the secret from multiple shares: it takes a set of data objects as input, and produces as output the result of the reconstruction. The number of input elements should be greater than or equal to the threshold value set in the *SSSsharing* stereotyped task.

Besides the standard secret sharing, it is possible to use two specialisations: *additive secret sharing* (Figure 6) and *function secret sharing* (Figure 7). In the additive secret sharing, the number of shares or computation

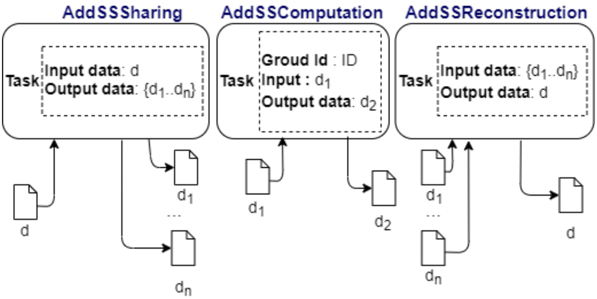


Figure 6: Additive secret sharing.

shares needed to reconstruct the secret is equal to the number of shares created when the secret has been split. Thus, differently from the standard secret sharing, the *AddSSSharing* task does not specify the threshold and the number of computation parties.

This type of task works in combination with the *AddSSComputation* and *AddSSReconstruction* tasks, which keep the same attributes of *SSComputation* and *SSReconstruction*, respectively.

Instead, in function secret sharing the secrets, as well as the shares, are expressed in the form of functions [104]. The number of shares is fixed to two, and as a consequence also the threshold to reconstruct the secret is fixed to two.

A *FunSSSharing* task produces two additive shares from a data object. A *FunSSComputation* task takes in input a share and an evaluation point (i.e., the value over which the function is executed), and generates in output a share. Also in this case the group id attribute is used to identify all the tasks running the same computation over different shares. A *FunSSReconstruction* task takes the two input shares and gives in output the result of the reconstruction.

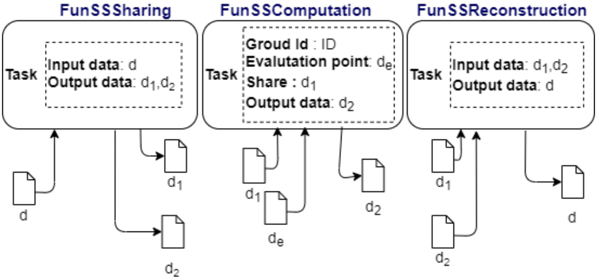


Figure 7: Function secret sharing.

Encryption. This PET uses a key to cipher the data objects to protect. The data objects can be deciphered only by using the correct key. We consider two main types of encryption: *public key* and *symmetric key encryption*.

The *public key encryption* in Figure 8 uses a public key to cipher the secret data, and a private key to decipher it. The keys are two essential elements of the model: the one used to cipher the data is annotated with the stereotype *PKPublic*, while the one used to decipher the ciphertext, i.e. ciphered data, is annotated with *PKPrivate*.

A *PKEncrypt* task ciphers the input data with the key, and produces in output the ciphertext. A *PKComputation* task executes some computa-

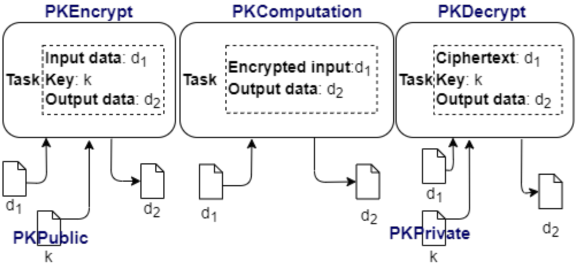


Figure 8: Public Key Encryption.

tion over the input ciphertext, and produces another ciphertext as result. A *PKDecrypt* task uses the key to decrypt the ciphertext, and gives the corresponding plaintext back.

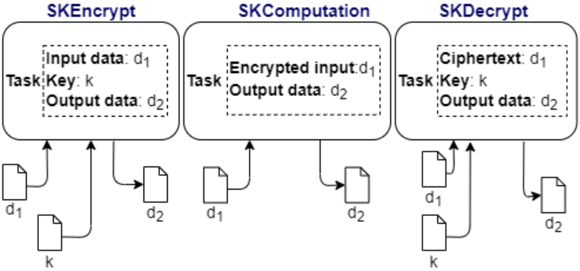


Figure 9: Symmetric key encryption.

In the *symmetric key encryption*, shown in Figure 9, we lose the concept of the public and private key. Instead, we use one key to both cipher and decipher the input data object and ciphertext respectively. The stereotypes used to implement it are *SKEncrypt*, *SKComputation* and *SKDecrypt*. All the attributes of the stereotyped tasks are the same of the tasks for public key encryption.

Multi-party computation (MPC). This PET, shown in Figure 10, allows a set of parties to compute a function over their inputs while keeping these inputs private [41]. MPC requires the collaboration of multiple independent participants of the same group during the computation [104], which means that the execution is synchronous.

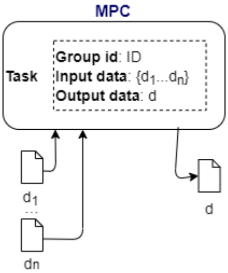


Figure 10: Multiparty computation.

To consider how PE-BPMN works in practice, we enhanced the model in Figure 2 with data objects and secret sharing PET stereotypes. The formalisation in Chapter (3) equates intermediate throw and catch events with their corresponding tasks, which means that we can substitute them with ″send data″, ″receive data″, ″send results″, and ″receive results″ tasks. Notably, this replacement does not affect the behaviour of event-based gateway. In fact, the BPMN notation allows to connect receiving tasks to the event-based gateway (see [95, Section 10.5.6]); this fits well with the replacement of intermediate events with tasks we perform in our approach.

The resulting PE-BPMN model is shown in Figure 11, where for the sake of readability we omitted the tasks' attributes (we clarify how they are set up below).

The *SSSharing* stereotype annotates the task $S1$, setting the value of the threshold and computation parties to 2. $S1$ generates $share1$ and $share2$, and sends the latter one to the Data Centre party via the $share$ message. This will use it for the computation during the execution of the task $D2$, and will send back the result of the computation in the message $result$. The Space Agency takes part in the computation through the task $S3$. Notably, tasks $S3$ and $D2$, which are annotated with the stereotype *SSComputation*, take part to the same group id. In the end, task $S7$, which has the *SSReconstruction* stereotype, puts together the shares $result1$ and

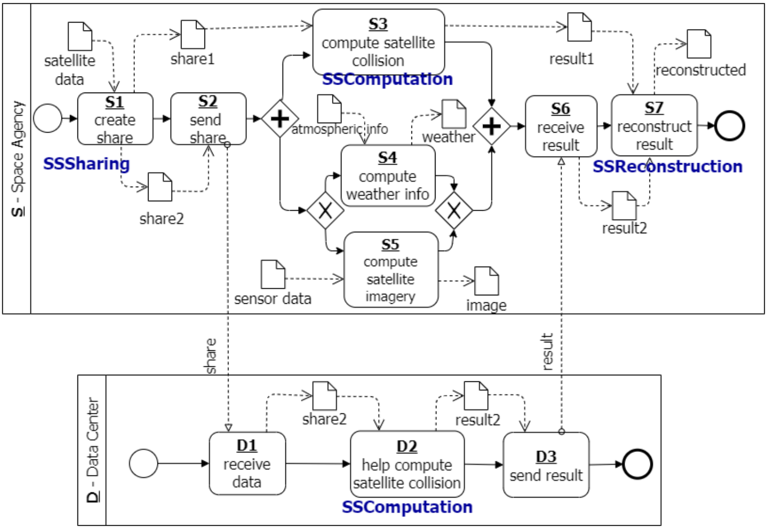


Figure 11: PE-BPMN model enhanced with data objects and secret sharing.

$result2$ to obtain the $reconstructed$ result.

The Data Centre party will never receive $share1$ or $result1$, while the Space Agency has the right to access both the secrets because it is the party that generated the shares via the task $S1$ and also is the party in charge of reconstructing the result.

Pleak is a modelling and analysis environment for privacy-enhanced systems. It gives the possibility to model privacy-enhanced systems and provides privacy audit features, like sensitive or guessing advantage analysis [98, 125, 47].

Pleak already provided some means to verify PE-BPMN models. For example, check that each task with privacy stereotypes has the right inputs and outputs. By enabling formal verification through the top-down approach, we demonstrate more complex properties highlighted in Section 3.1, like the possibility of reconstructing protected data (violating the defined PET) or analysing branching.

2.3 Process Mining

Process Mining is a discipline combining data mining, computational intelligence and process modelling and analysis [2]. Process mining aims at extracting useful information from event logs for discovering, monitoring and improving real processes [3]. It is an evidence-based approach and ensures a close correspondence between modelled and observed behaviour, given that the model's definition is based on actual process execution traces.

The underlining element of process mining are the event logs. An **event log** is a set of cases, while a case is a sequence of events with attributes that indicate activity name, time, cost, used resource, etc. Event logs are usually formatted using the eXtensible Event Stream (XES) standard format [58].

Process mining consists of three different techniques: *discovery*, to produce a model from an event log without using any a priori information, *conformance*, used to check if the model is correct concerning the log, and *enhancement*, to improve existing process model using information about the actual process recorded in some event log.

Our work takes inspiration from a process discovery technique to provide an automatic way of generating models while enabling formal verification capabilities. Following, we present some of the best process discovery techniques used later to validate our bottom-up approach.

2.3.1 Process Discovery Techniques

Process discovery techniques are algorithms able to produce models from an event log automatically. This method can be used to study the control-flow perspective of a process in an offline setting, i.e. when a case is already completed. Over the years, several mining algorithms have been developed [14], which differ in the kind and quality of the output.

The *Schimm algorithm* [115] is the algorithm from which our bottom-up approach is inspired. For this reason, we will present it followed by the three mining algorithms used to validate the obtained results, that are: the *Structured Heuristics Miner* [12] first discovers models that are possibly unstructured and unsound and then transforms them into structured and sound ones; the *Inductive Miner* [77] extracts block-structured process models, called process trees, by splitting the logs into smaller pieces and combining them with tree operators; and, finally, the *Split Miner* [13] aims at identifying a combination of split gateways to capture behaviours like concurrency or casual relations while filtering the graphs derived by the event logs.

Schimm algorithm [115]. This algorithm aims to mine exact workflow models. Exact means that the model respect *completeness* (all tasks and dependencies present in the log are maintained), *specificity* (the model does not introduce additional tasks) and *minimality* (minimal number of elements to describe the workflow) [115]. This technique depends strongly on which type of log is given in input. It has to be an event log where, for each activity in a trace, the start and complete time are always declared; otherwise, it cannot spot parallel execution. Another characteristic is that it does not consider measures like the frequency of an event, meaning that the generated model repeats every single action recorded in the log. Taking into account infrequent behaviour can be problematic if the aim is to highlight the process capabilities. Still, it could be a strength when the goal is to identify unexpected executions. Moreover, the log has to be complete and noise-free; otherwise, errors will be included in the model.

The output structure is a *block-oriented metamodel*. This kind of model is well-formed, safe and sound (no deadlocks, always has the option to complete). This model uses operators (Sequence, Parallel, Alternative and Loop) to combine blocks and define the control flow. The compositional capability of process algebra makes it easy to reproduce the block structure behaviour.

Structured Heuristic Miner [12]. This is an improved version of the heuristic miner presented in [136]. This algorithm aims to find BPMN models providing the right tradeoff between accuracy, soundness and structured models, often more understandable than unstructured ones.

The accuracy tries to maximise the three main criteria used in process mining to measure models, i.e. *fitness*, models ability to reproduce the traces of the log, *precision*, behaviour shown by the model but not included in the log, and *generalisation*, capability of producing a trace that is not in the log but is part of the process.

The structured heuristic miner first discovers a model using the heuristic algorithm capable of producing unstructured and unsound models, then transforms it into a structured and sound one. However, there are cases where the heuristic phase leads to an imprecise or spaghetti-like model, and the structuring phase cannot fully structure the model and repair its unsoundness.

Inductive Miner [77]. It aims to discover models based on the Pareto principle, which states that 20% of the model is enough to represent 80% of the total behaviour of the system [78]. This technique can filter log *infrequent behaviour* while providing a sound model. This is based, as the Schimm algorithm, on block-structured models called process trees. In a *process tree*, the leaf represents activities, while non-leaf nodes contain operators that define the relationships among their children.

This algorithm can improve the model's fitness and is an excellent choice to abstract from the log to capture the overall behaviour. For example, it can be applied for the identification of social networks that do not require a high precision value, while is not suitable to find deadlocks or spot unexpected system behaviour.

Split Miner [13]. It is one of the newest algorithms developed in the field of process discovery. It tries to generate sound models with low branching complexity in a faster way and tries to have a high and balanced fitness, precision and generalisation value [13].

The output is a *sound BPMN model* with no deadlock, satisfying option to complete (every task has a path from the start to the end point) and proper completion (no branches are active at the end of the execution). This last property is not guaranteed in the case of cyclic process models, because they may generate more than one instance of execution.

Chapter 3

From Collaboration Models to Formal Specifications

This chapter presents the *top-down approach* that bridges the gap between BPM and FM by defining a formalisation of the BPMN models to enable formal verification. Verification is essential when dealing with models enhanced with data that need to consider how this data is handled internally and especially among different organisations.

To this end, we consider collaborative systems that meet strict privacy requirements designed through standards that extend existing modelling languages to capture privacy aspects [41]. Accordingly, the thesis considers **PE-BPMN** [104, 105], an extension of BPMN designed to capture *privacy-enhancing mechanisms* in collaborative business processes.

While PE-BPMN allows analysts to specify privacy mechanisms and does some analysis, some techniques still need to be added to verify that the resulting privacy-enhanced process models fulfil the intended privacy properties. In particular, methods and tools are missing for detecting privacy leakages in privacy-enhanced collaborative processes [5], i.e., states where a peer in the collaborative process may unduly gain access to private information.

Our top-down approach provides a **verification methodology for PE-**

BPMN models to detect different types of privacy leakages that may occur during the execution of PE-BPMN models due to mistakes in the design phase or the incorrect usage of privacy-enhancing technologies. Of course, since PE-BPMN is a conservative extension of BPMN, the approach we propose also works with standard BPMN models for verifying more traditional properties of business processes (e.g., absence of deadlock)

The privacy-enhancing technologies considered, also presented in Section 2.2.1, are the *secret sharing technology* and two of its specialisations, *additive* and *function secret sharing*. Secret sharing allows a secret to be divided between different parts so that the secret can be reconstructed by combining all or some of the parts. This technology is used to exchange and store highly sensitive information. In this setting, it is crucial to ensure that the parties cannot reconstruct the secret without having the right to do so.

Another privacy-enhancing technology is encryption. PE-BPMN supports the specification of processes that use both *public-key* and *symmetric-key encryption*. In this approach, a secret is encoded using a key and can be decrypted only using the same or a corresponding key. In this thesis, we address the problem of checking that no party gets access to the secret key without authorisation.

Furthermore, the thesis considers the problem of verifying the correct design of *multi-party computations*, which allows independent parties to collaboratively compute a function on the data they hold while keeping these data private by checking that the computation is executed synchronously.

To allow these verification capabilities, we rely on a **formalisation of PE-BPMN in terms of the process algebra mCRL2** [27]. We use a process algebraic approach to take advantage of its intrinsic compositional nature. This approach is particularly effective in collaboration models, as the behaviour of the distributed parties can be specified separately and then composed to capture the overall collaboration behaviour.

The choice to use $mCRL2$ is motivated by: *(i)* the suitability of the operators and features provided by this formalism to capture control-flow, messages-flow, and data aspects of PE-BPMN models; and *(ii)* the availability of the advanced model checking capabilities in the $mCRL2$ toolset as discussed in Section 2.1.

To validate the feasibility and applicability of the proposed approach, we have implemented it as a tool that takes as input a PE-BPMN model, computes its corresponding $mCRL2$ specification, and checks for data leakages or errors in the design of the privacy features. The tool allows users to choose which privacy-related properties to verify on the PE-BPMN model specification. The verification output includes an example pathway illustrating each detected privacy leakage or misconfiguration.

We have integrated this verification prototype as a plugin within the Pleak privacy analysis toolset [98, 125, 47]. We have captured and analysed several realistic collaborative business processes using this plugin.

The rest of the Chapter is organised as follows: Section 3.1 provides a general description of the methodology, Section 3.2 explains the formalisation from PE-BPMN to $mCRL2$ formal specification, Section 3.3 shows in detail the verification capability provided and how they are embedded in the formal specification, Section 3.4 and Section 3.5 presents the tool implementation and the validation activities executed respectively. Finally, Section 3.6 discusses some related work about the current BPMN formalisation and verification of privacy-related properties methodology.

3.1 A methodology for PE-BPMN Collaborations Verification

In this section, we provide an overview of the approach we use to verify PET-related properties over PE-BPMN models.

The input model of our verification methodology is a PE-BPMN collaboration diagram. To simplify the formal treatment, we make two assumptions on the input model:

(i) the model is *well-structured* [69] (also known as *block-structured*), imposing gateways in each process to form single-entry-single-exit fragments

(ii) each task can send/receive at most one message

The first assumption is not overly restrictive, as it has been shown in previous works that a large class of process models can be re-written as block-structured process models [100].

The second assumption comes without loss of generality, as it is always possible to safely transform a complex task with multiple outgoing/incoming message flows in a sequence of separate tasks, each of which with at most one message flow, with exactly the same meaning. This simplification is also aligned with generally accepted modelling guidelines [86, 34]. In fact, it helps to avoid misunderstandings in the execution order among the send/receive actions performed within a task, thus allowing the designer to get a clear understanding of what is happening in the model execution.

The methodology, represented in Figure 12, consists of four steps:

1. *Control-flow transformation*,

2. *PETs identification*,

3. *Data- and message-flow transformation*,

4. *Verification*.

In the first step, the process in each pool of the PE-BPMN model is transformed into a *process algebra specification*. This step focus on the control-flow perspective of the model, i.e. we abstract from PETs, data objects and message flows. The data-abstracted structure of each process is represented as a *process tree*, which is an intermediate representation that can be then easily transformed into a $mCRL2$ process specification.

In the second step, we extract information concerning *PET stereotype annotations* from the PE-BPMN model. This information allows us to properly deal with PETs in the subsequent steps.

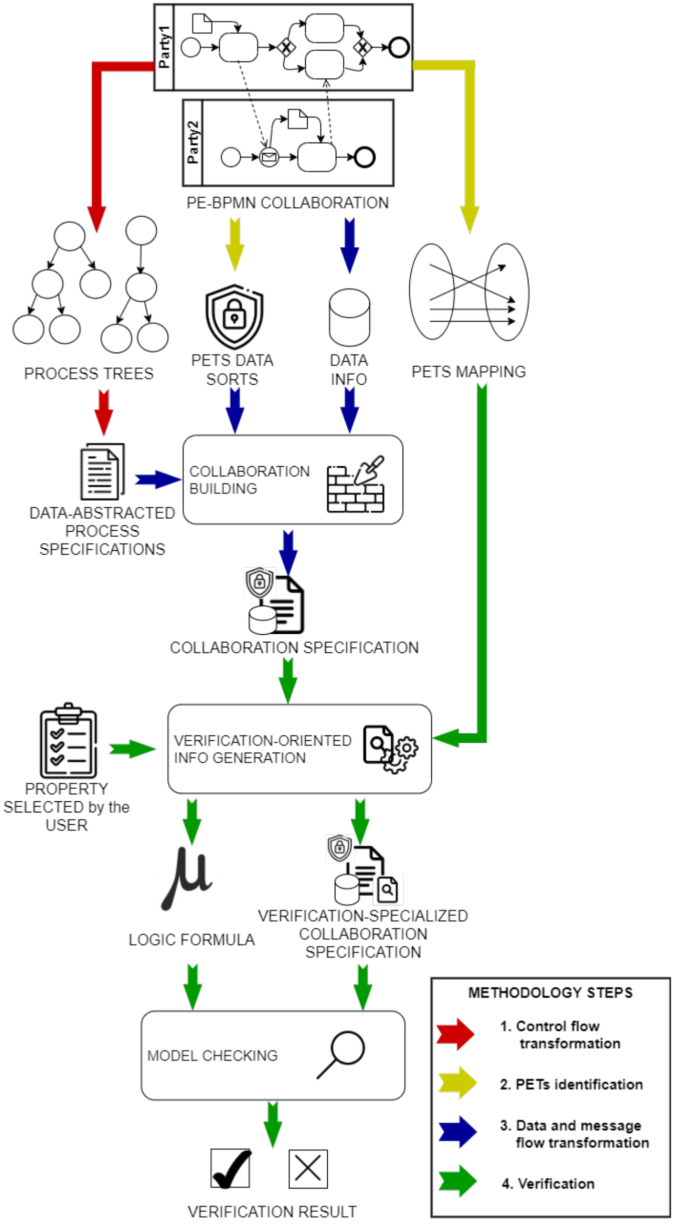

Party1
Party2
PE-BPMN COLLABORATION
PROCESS TREES
PETS DATA SORTS
DATA INFO
PETS MAPPING
DATA-ABSTRACTED PROCESS SPECIFICATIONS
COLLABORATION BUILDING
COLLABORATION SPECIFICATION
PROPERTY SELECTED by the USER
VERIFICATION-ORIENTED INFO GENERATION
LOGIC FORMULA
VERIFICATION-SPECIALIZED COLLABORATION SPECIFICATION
MODEL CHECKING
VERIFICATION RESULT
METHODOLOGY STEPS
1. Control flow transformation
2. PETs identification
3. Data and message flow transformation
4. Verification

Figure 12: Overview of the approach

In the third step, the specification of each task is enhanced to capture interactions with data objects and exchange of messages, while each data and message communication is encoded via a *buffer*. The generated terms for processes and data handling are then combined via parallel composition, resulting in an overall data-aware specification of the collaboration.

The fourth step focus on verifying a set of privacy-related properties chosen by the user against the $mCRL2$ specification. For each of them, we obtain a *violation* or *no-violation* answer and, in the former case, a counterexample is generated.

Given a collaboration specification, the properties that the user can verify are as follows.

- **Task verification.** *Can a task read a set of data?* In other words, is there a path in the model execution leading to a state where the content of the considered data objects is part of the task's knowledge?

- **Participant verification.** *Can a participant read a set of data?* In other words, is there a reachable state in which the participant has knowledge of every element in the set of data?

- **Secret sharing/Additive secret sharing/Function secret sharing verification.** *Can a participant get to know a number of secret shares or computed shares greater than or equal to the threshold number set to reconstruct it without having the right to do it?* In other words, is there a path leading to a state where the participant that has not created the shares (no SSSharing/AddSSSharing/FunSSSharing task) nor reconstructed them (no SSReconstruction/AddReconstruction/Fun-Reconstruction task) has knowledge of a number n of shares or computed shares enough to reconstruct the secret (i.e., $n \geq t$, where t is the threshold)?

- **Private key/Symmetric key encryption verification.** *Can a participant get to know the cipher and its decoding key without being qualified to have it?* In other words, is there a path leading to a state where the participant that has not created the cipher (no PKEncrypt/SKEncrypt task) nor decrypted it (no PKDecrypt/SKDecrypt task) reads the cipher and its decoding key?

- **Reconstruction verification.** *In a secret sharing scenario, can a task with the right to reconstruct the secret not have enough or computed shares to reconstruct it?* In other words, in a model with secret sharing/additive secret sharing/function secret sharing can a path exist such that it leads to a state where a task enhanced with the SSReconstruction/AddSSRecontruction/FunSSReconstruction stereotype has a number of shares or computed shares less than the threshold number to reconstruct the initial secret?

- **MPC Verification.** *Can a group of MPC tasks not be executed synchronously?* In other words, is there a path where the execution of a task cannot start until the execution of another task ends, while the two tasks are in the same MPC group?

- **Deadlock freedom.** *Is there a participant in the collaboration who is not able to finish its execution?* In other words, is there a path leading to a state where not all the end events have been executed?

In the case of *task* and *participant verification*, if the property is satisfied it means that no violation occurred in the model; for all the other types of verification, instead, the satisfaction of the property means that a violation exists.

Notably, when the property to verify is selected by the user, we generate the $mCRL2$ elements corresponding to the chosen property and we embed it inside the specification, by adding maps, variables and new processes depending on the encoding of the property. After the verification, the specification is cleaned from this verification-oriented information in order to be possibly enhanced again with new information for the verification of another property.

3.2 From PE-BPMN to $mCRL2$

This section illustrates how to transform a PE-BPMN collaboration model into a $mCRL2$ specification. This transformation results from the execution of the first three steps of the methodology presented above, which are detailed in Sections 3.2.1, 3.2.2 and 3.2.3, respectively.

3.2.1 Control-flow Transformation

This step aims at generating a coarse-grained specification that consider only the *control-flow structure* of the PE-BPMN model. To simplify the formal definition of the transformation, as well as its implementation, we resort to a tree-based representation of PE-BPMN models. In particular, we have defined a structure, called **process tree**, which is a variant of the RPST (Refined Process Structure Tree) introduced in [132, 101].

Definition 3.2.1 (Process Trees)**.** *The syntax of process trees is as follows.*

$$\begin{aligned} t ::= \; & start(id) \;\mid\; end(id) \;\mid\; task(id) \;\mid\; seq(t_1, ..., t_n) \\ & \mid\; xor(t_1, ..., t_n) \;\mid\; and(t_1, ..., t_n) \;\mid\; ebg(t_1, ...t_n) \;\mid\; while(t) \end{aligned}$$

where id denotes a unique element identifier[1]*.*

The correspondence between the graphical representation of a PE-BPMN model and its process tree representation is straightforward, as shown in Table 1. Since the formalism equates throw and catch events with tasks, their notation is not present in Table 1.

The generation of the process tree corresponding to a process of a PE-BPMN collaboration is significantly simplified by the well-structuredness assumption. We refer to the literature about RPST, in particular to [101], for details on the procedure for the generation of the tree-based representation.

For the sake of presentation, in the examples included in this section, we use the name of the organisation for identifying start and end elements (since every process has just one start and one end), and we use

[1] Notably, although these identifiers are not explicitly reported in the graphical representation of BPMN models, they are reported in the XML representations of the models, as prescribed by the BPMN standard.

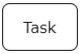
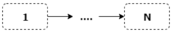
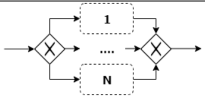
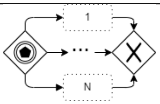
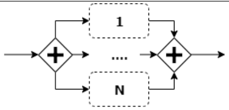
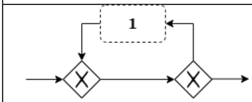
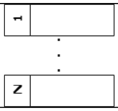
PE-BPMN Element	Process tree
	$start(id)$
	$end(id)$
	$task(id)$
	$seq(t_1, \ldots, t_N)$
	$xor(t_1, \ldots, t_N)$
	$ebg(t_1, ..., t_N)$
	$and(t_1, \ldots, t_N)$
	$while(t_1)$
	$t_1 \ \ldots \ t_N$

Table 1: Correspondence between the PE-BPMN block-structures and process tree elements.

the task name for identifying tasks.

Let us consider the collaboration model in Figure 11. The process trees corresponding to the processes of the two parties, which are graphically depicted in Figure 13, are as follows:

$$\begin{aligned} \mathsf{S}: \quad & seq(start(S), task(S1), task(S2), and(task(S3), xor(task(S4), \\ & \quad task(S5))), task(S6), task(S7), end(S)) \\ \mathsf{D}: \quad & seq(start(D), task(D1), task(D2), task(D3), end(D) \end{aligned}$$

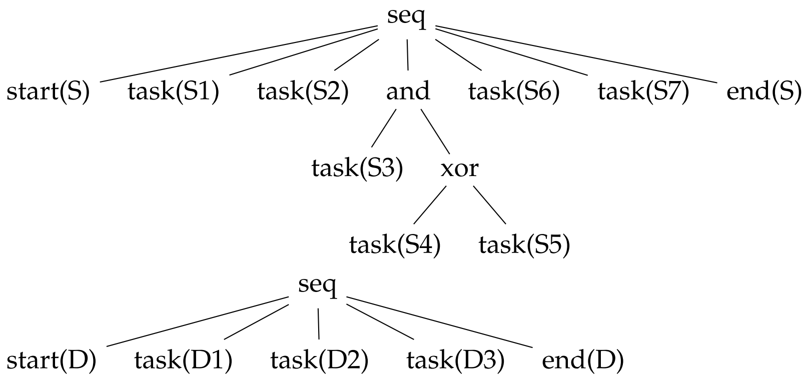


Figure 13: Process trees of parties S and D from Figure 11.

We can notice that there is a direct correspondence between tree nodes and (blocks of) elements in the PE-BPMN model. Notably, while the children order does not matter for and and xor nodes, it is relevant for seq nodes (as one may expect, the execution order is from left to right).

We can now formalise the control-flow transformation step by means of the **translation function** $\mathcal{T} : \mathbb{P} \to \mathbb{M}$, where $\mathbb{P}$ is the set of process trees and $\mathbb{M}$ the set of mCRL2 terms.

Definition 3.2.2 (Translation function)**.** *Function $\mathcal{T}$ is inductively defined as follows:*

$$\begin{array}{lcl}
\mathcal{T}(start(id)) & = & start\_id(\{\ \}) \\
\mathcal{T}(end(id)) & = & end\_id(\{\ \}) \\
\mathcal{T}(task(id)) & = & id(\{\ \}) \\
\mathcal{T}(seq(t_1,\ldots,t_n)) & = & \mathcal{T}(t_1).\cdots.\mathcal{T}(t_n) \\
\mathcal{T}(xor(t_1,\ldots,t_n)) & = & tau.\mathcal{T}(t_1) + \cdots + tau.\mathcal{T}(t_n) \\
\mathcal{T}(ebg(t_1,\ldots,t_n)) & = & \mathcal{T}(t_1) + \cdots + \mathcal{T}(t_n) \\
\mathcal{T}(and(t_1,\ldots,t_n)) & = & \mathcal{T}(t_1) \,||\, \cdots \,||\, \mathcal{T}(t_n)
\end{array}$$

$$\mathcal{T}(while(t)) = K \quad \textit{with } K = tau\ + tau.\mathcal{T}(t).K \ \ K \textit{ fresh}$$

We comment on salient points. *Start*, *end* and *task* elements are translated into visible actions with empty knowledge. The knowledge of tasks will be enriched during the data/message flow transformation step for those tasks that exchange information at some stage of the process.

Sequence and *parallel* blocks are expressed by means of sequential and interleaving operators, respectively. Both *exclusive* and *event-based* blocks use the choice operator: in the former case the non-deterministic selection is internal, due to the use of the tau action (i.e., silent action) as prefix, while in the latter case the selection is driven by the initial actions of the block branches. The *while* block is rendered as a call of a recursive process, identified by a fresh identifier. At each call, the process non-deterministically decides whether to stop the loop or to execute the body and restart.

For example, the processes included in the PE-BPMN collaboration model in Figure 11 are transformed into the following data-abstracted process specifications:

$$start\_S(\{\,\})\,.\,S1(\{\,\})\,.\,S2(\{\,\})\,.(S3(\{\,\})\;||\;(tau.S4(\{\,\})\,+\,tau.S5(\{\,\})))\,.\\ S6(\{\,\})\,.\,S7(\{\,\})\,.\,end\_S(\{\,\})$$

$$start\_D(\{\,\})\,.\,D1(\{\,\})\,.\,D2(\{\,\})\,.\,D3(\{\,\})\,.\,end\_D(\{\,\})$$

Although the transformation introduced above produces legal $mCRL2$ specifications, unfortunately, not all of them are accepted as input by the $mCRL2$ supporting tools.

Indeed, the interleaving operator can only be used on the outer level of a specification (see [84] for more details). Anyway, this limitation can easily be overcome by applying the $\mathcal{T}_p$ function defined later in Section 4.2, in order to move all nested parallel compositions to the outer level.

As a matter of example, the term $a.(b||c).d$ will be transformed in the equivalent term $hide(\{t'\}, allow(\{a, b, c, d, t'\}, comm(\{t|t{-}{>}t'\},\ (a.t.b.t.d\ ||\ t.c.t))))$, where the original order imposed by the process is preserved by means of actions t used as synchronisation points, which produce actions t' that are hidden.

3.2.2 PETs Identification

The identification of PETs is essential to correctly replicate the behaviour of the tasks associated to them. As shown in Figure 12, this step of the methodology produces two kinds of information: data sort definitions and a function mapping BPMN elements to PET stereotypes.

The *sort definitions* are necessary, in the data transformation step, to enrich the $mCRL2$ specification with technical details for recognising those elements that are part of privacy technologies and, hence, properly dealing with them. This specification enhancement together with the mapping function give us, in the verification step, the possibility to define functions to check if PETs are violated or not.

Definition 3.2.3 (Sort definitions)**.** *Given a PE-BPMN collaboration model, the sort definitions produced in the PETs identification step are as follows:*

$$
\begin{aligned}
\mathit{sort\ Data} \ &= \ \mathit{struct\ node}(value : Name)?is\_node \ | \\
&\qquad\qquad pnode(pvalue : Privacy)?is\_pnode \ | \\
&\qquad\qquad eps \ | \ vnull \\
Name \ &= \ \mathit{struct}\ data_1 \ | \ \ldots \ | \ data_n \\
Privacy \ &= \ \mathit{struct}\ pair(frt : PName, snd : Nat) \\
Pname \ &= \ \mathit{struct}\ pname_1(Name)?is\_pname_1 \\
&\qquad\qquad | \ \ldots \ | \\
&\qquad\qquad pname_m(Name)?is\_pname_m
\end{aligned}
$$

where $data_1 \ldots data_n$ are the names of the data objects that are specified in the PE-BPMN collaboration model, and $pname_1 \ldots pname_m$ are the privacy features used in the PE-BPMN collaboration model.

$Data$ is a structured sort, whose elements are explicitly characterised as follows.

- $node$ represents a data object with no-privacy features. Its projection function, $value$, allows to extract the name of the data object, which is an element of the sort $Name$. As an example, given the PE-BPMN collaboration model in Figure 11, the data object names $data_i$ belonging to the sort $Name$ are $satellite\ data$, $share1$,

$share2$, $atmospheric\ info, \ldots$ The recogniser function, is_node, returns true if it is applied to a $node$ term.

- $pnode$ represents a data object enhanced with a privacy feature and its projection function, $pvalue$, extracts a data of sort $Privacy$. A term of the $Privacy$ sort denotes a $pair(frt, snd)$, where frt is an element of sort $PName$ and snd its a natural number. $PName$ is the sort that represents the privacy features in the specification. As an example, in the PE-BPMN collaboration model in Figure 11 the privacy features $pname_j$ belonging to the sort $PName$ are $SSSharing$, $SSComputation$, and $SSReconstruction$. The natural number in the pair corresponds to an id, whose meaning changes on the basis of the $PName$ associated to it.

- eps represents the non-presence of data.

- $vnull$ represents the data with value $null$.

Definition 3.2.4 (Mapping function). *Given a PE-BPMN collaboration model, the mapping function pet generated in the PETs identification step is a partial function from task identifiers and data object names to PETs stereotypes as specified in the PETs annotations of the model.*

For example, in the PE-BPMN collaboration model in Figure 11, the function pet is defined as follows[2]: $pet(S1) = SSSharing$, $pet(S3) = SSComputation, pet(S7) = SSReconstruction, pet(D2) = SSComputation$, and pet is undefined for all other identifiers.

3.2.3 Data- and Message-flow Transformation

In the previous steps of the proposed methodology, we described how to formalise the control flow of each process involved in the collaboration model, and the data definitions necessary for bringing the $mCRL2$ specification to fruition.

[2] As usual, for the sake of presentation, we use here task names in place of task identifiers.

In this step, we show how all pieces of the puzzle fall into place. First, we combine the data-abstracted process specifications together, then we enrich the specification in order to deal with data and message handling.

Definition 3.2.5 (Collaboration building). *Given a collaboration involving n parties, let t_1, ..., t_n be the process trees generated from each of them, then the overall specification can be defined as:*

$$
\begin{array}{l}
sort\ Data \ = \ struct\ node(value : Name)?is\_node \mid \ldots \\
\quad\quad Name \ = \ struct\ data_1 \mid \ldots \\
\quad\quad \ldots \\
K_{party\_1} = \mathcal{T}(t_1);\ \ \ldots\ \ K_{party\_n} = \mathcal{T}(t_n); \\
init\ (K_{party\_1} \mid\mid \ldots \mid\mid K_{party\_n})
\end{array}
$$

where $K_{party\_1}, \ldots, K_{party\_n}$ are fresh identifiers, and $init$ is a mCRL2 keyword that defines the initial behaviour.

For example, the PE-BPMN collaboration model in Figure 11 is transformed into the following $mCRL2$ term providing an overall specification of the control flow of the model:

$$
\begin{array}{l}
sort\ Data\ = \ldots \\
K_S \ = \ start\_S(\{\,\}) \,.\, S1(\{\,\}) \,.\, \ldots \,.\, end\_S(\{\,\}) \\
K_D \ = \ start\_D(\{\,\}) \,.\, D1(\{\,\}) \,.\, \ldots \,.\, end\_D(\{\,\}) \\
init\ (K_S \mid\mid K_D)
\end{array}
$$

Let us focus now on the transformation of data and message flows. In PE-BPMN the exchange of data is asynchronous and can take place either *intra-pool*, via data-object connections between tasks of the same process, or *inter-pool*, via message flows connecting tasks of separate processes.

Both forms of communication rely on buffers and a non-blocking sending task. They differ, instead, on the behaviour of the receiving task: in the inter-pool interaction, the receive is blocking if the buffer is empty, while in the intra-pool one, the execution of the receiving task can continue as an empty message will be received. These two behaviours are captured through two buffer specifications.

In addition, a task can have incoming data-objects that are not produced by other tasks; the information they bring is called *prior knowledge*. This information is hence directly inserted inside the task specification.

Therefore, the PE-BPMN model is analysed to extract all information concerning communication, which is then used to enrich the $mCRL2$ specifications produced in the previous step. We illustrate below how the task specifications are enhanced with data information and how the two kinds of buffers are defined.

Definition 3.2.6 (Data-aware specification of tasks). *Given a task with id as identifier, j incoming data links/message flows, r outgoing data links/message flows, and data $e'_1, ..., e'_p$ as prior knowledge, its mCRL2 specification becomes the following one:*

$$\begin{aligned}
&sum\ e_{11}, ..., e_{1k_1} : Data.i_1(e_{11}, ..., e_{1k_1}).\\
&... .\\
&sum\ e_{j1}, ..., e_{jk_j} : Data.i_j(e_{j1}, ..., e_{jk_j}).\\
&id(\{e'_1, ..., e'_p, e_{11}, ..., e_{1k_1}, ..., e_{j1}, ..., e_{jk_j}\}).\\
&o_1(e''_{11}, ..., e''_{1h_1}). ... .o_r(e''_{r1}, ..., e''_{rh_r})
\end{aligned}$$

where the sort $Data$ is defined in Def. 3.2.3.

Using the . operator among incoming/outgoing message flows we impose an arbitrary order among how a task is receiving/sending the data that is not given in the PE-BPMN model. This decision does not really affect the behaviour of the specification, as we will see later, because the result of this interactions are going to be hidden in the final specification.

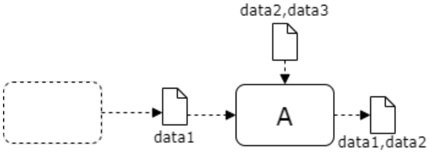


Figure 14: Example of a task.

As an example, let us consider the task in Figure 14; its data-aware translation is as follows:

$$\begin{aligned}
&sum\ e_1 : Data.i(e_1).\\
&\qquad A\big(\{node(data2), node(data3), e_1\}\big).\\
&\qquad\qquad o(e_1, node(data2))
\end{aligned}$$

where: e_1 is a placeholder for a data parameter of type $Data$ to be received through the input action i (i.e. $data1$); the input parameters are data objects coming from other tasks. The knowledge of the task then consists of the data that is received (i.e., e_1 placeholder for $data1$) and the prior knowledge data (i.e., $data2$ and $data3$) which are data objects that the task is not receiving from another task; and the output data (i.e., e_1 and $data2$) is transmitted via action o.

Every communication between two tasks internal to a pool is realised by means of a dedicated buffer.

Definition 3.2.7 (Intra-communication buffer)**.** *The intra-communication buffer is a process of the following form:*

$$\begin{aligned} B(d_1, ..., d_n : Data) = \quad & sum\ e_1, ..., e_n : Data.\ i(e_1, ..., e_n).B(e_1, ..., e_n) \\ & + o(d_1, ..., d_n).B(vnull, ..., vnull) \end{aligned}$$

where B is a fresh name for a process with n parameters, i the input channel for writing in the buffer and o the output channel for reading from it.

Notably, the buffer is defined as a recursive process in order to deal with more than one communication in case of loops in the model. Every intra-communication buffer is put in parallel with the other processes at top level of the specification, and is initialised as $B(eps, ..., eps)$, where eps represents the empty parameter.

This is indeed a non-blocking buffer: if no data is written in the buffer, it provides a null output ($vnull$). Notice that, for the sake of simplicity, we have used a 1-position buffer that, each time it receives new data, it rewrites the current one.

Definition 3.2.8 (Intra-communication protocol)**.** *In a collaboration with k participants, let $task(id_1)$ and $task(id_2)$ be tasks in the same pool such that the former is sending a set of data $D = \{d_1, \ldots, d_n\}$ to the latter. Then, the $mCRL2$ processes corresponding to the tasks and to the intra-communication buffer are defined as follows:*

$$\begin{aligned} & K_1 = id_1(\{d_1, \ldots, d_n\}).o_1(d_1, \ldots, d_n) \\ & K_2 = sum\ e_1, \ldots, e_n : Data.i_2(\{e_1, \ldots, e_n\}).id_2(\{e_1, \ldots, e_n\}) \\ & B(d_1, \ldots, d_n : Data) = sum\ e_1, \ldots, e_n : Data.i_(e_1, ..., e_n).B(e_1, ..., e_n) \\ & \qquad\qquad\qquad\qquad\quad + o_b(d_1, ..., d_n).B(vnull, ..., vnull) \end{aligned}$$

Then the communication between these elements is specified as follows:

$$\begin{aligned}&init\ hide(\{sr\}, allow(\{sr, id_1, id_2\} \cup Act,\\ &\qquad\qquad comm(\{o_1|i_2{-}{>}sr\}, P_1||P_2||...||P_k||B(eps, \ldots, eps))))\end{aligned}$$

where sr it is an action used to represent the result of the communication and $P_1, P_2, \ldots, P_k$ are the processes representing the parties in the collaboration.

According to the above definition, for every communication we will have a buffer process B that is part of the initial behaviour of the specification, in order to receive at any time in the execution a set of data.

A communication function ($o_1|i_2{-}{>}sr$) between the output channel of the sending task (o_1) and the input channel of the receiving task (i_2) is generated, and their synchronisation is forced by allowing only the execution of the communication function.

Finally, since we are not interested in observing the sr synchronisation actions, they will be transformed into tau actions using an enclosing *hide* operator.

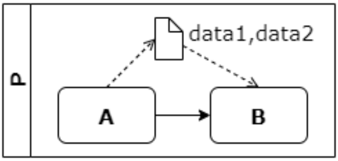


Figure 15: Example of intra-communication.

Let us now consider the minimal example in Figure 15 showing the role of the buffer. The communication between task A and task B is specified as follows:

$$\begin{array}{rcl}K_P & = & K_A.K_B\\ K_A & = & A(\{node(data1), node(data2)\}).\\ & & o_A(node(data1), node(data2))\\ B_{AB}(d_1, d_2) & = & sum\ e_1, e_2 : Data.i_{AB}(e_1, e_2).B_{AB}(e_1, e_2)\\ & & +\, o_{AB}(d_1, d_2).B_{AB}(vnull, vnull)\\ K_B & = & sum\ e_1, e_2 : Data.i_B(e_1, e_2).B(\{e_1, e_2\})\end{array}$$

$$
\begin{aligned}
&init\ hide(\{sr\}, allow(\{sr, A, B\},\\
&\qquad comm(\{o_A|i_{AB} \rightarrow sr, o_{AB}|i_B \rightarrow sr\},\\
&\qquad\qquad K_P \mid\mid B_{AB}(eps, eps))))
\end{aligned}
$$

Also every communication between two tasks not in the same pool has its own buffer.

Definition 3.2.9 (Inter-communication buffer). *A buffer for an inter-communication between two tasks is defined as follows:*

$$
\begin{aligned}
B(d_1, ..., d_n : Data) = {} & sum\ e_1, \ldots, e_n : Data.\\
& i(e_1, \ldots, e_n).B(e_1, ..., e_n)\\
& + (!empty(d_1)\ \&\ \ldots\ \&\ !empty(d_n))\\
& \rightarrow o(d_1, ..., d_n).B(eps_1, ..., eps_n))
\end{aligned}
$$

where $empty$ is a function that, given a parameter of type $Data$, returns $true$ when the parameter is empty (i.e., it is equal to eps), $false$ otherwise. The buffer is initialised again with empty data.

This is a blocking buffer, because when it is empty the output along o is not provided (due to the condition operator $Cond \rightarrow P$, meaning "if $Cond$ then do process P"), hence the receiving task has to wait.

Definition 3.2.10 (Inter-communication protocol). *In a collaboration with k participants, let $task(id_1)$ and $task(id_2)$ be two tasks in different pools such that the former is sending a set of data $D = \{d_1, \ldots, d_n\}$ to the latter. Then, the corresponding mCRL2 processes are defined as in Def. 3.2.8, but using Def. 3.2.9 for the inter-communication buffer specification.*

Hence, the only difference between "*intra*" and "*inter*" communication is the definition of the buffer. As an example, the communication between task A and task B in Figure 16 is specified as follows:

$$
\begin{aligned}
K_A &= A(\{data1\}).o_A(data1)\\
B_m(d_1) &= sum\ e_1 : Data.i_{AB}(e_1).B_m(e_1)\\
&\quad + (!empty(d_1)) \rightarrow o_{AB}(d_1).B_m(eps))\\
K_B &= sum\ e_1 : Data.i_B(e_1).B(\{e_1\})
\end{aligned}
$$

$$
\begin{aligned}
&init\ hide(\{sr\}, allow(\{sr, A, B\},\\
&\qquad comm(\{o_A|i_{AB} \rightarrow sr, o_{AB}|i_B \rightarrow sr\},\\
&\qquad\qquad K_A \mid\mid K_B \mid\mid B_m(eps))))
\end{aligned}
$$

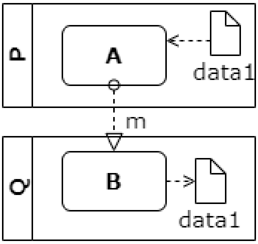


Figure 16: Example of inter-communication.

We conclude by illustrating the specification derived from our running example model (Figure 11) at the end of the three methodological steps introduced in this section. An excerpt of this specification is reported in Listing 3.1 where, for the sake of readability, we use the names of tasks and data objects inside the PE-BPMN model to define the actions of process tasks and data parameters, in place of the corresponding IDs. We describe below the specification blocks that compose the $mCRL2$ specification.

- $sort$ contains data type definitions. In this case, $PName$ reflects the fact that the PE-BPMN model uses secret sharing PETs stereotypes.
- map contains functions definitions. The function named $empty$ is used by buffers (Def. 3.2.7 and 3.2.9) to check if they received or not a value for a data object. This function takes as input a parameter of type $Data$ and gives as output a parameter of type $Bool$ ($Bool$ is a predefined sort in mCRL2 and contains the values $true$ and $false$).
- var defines the variable types used in eqn.
- eqn contains equations describing the behaviour of functions in map. In the example, the function $empty$ returns $true$, if the input parameter is equal to eps, otherwise $false$.
- act defines the action types. The actions of type $Collection$ are the PE-BPMN tasks, like $S1$ and $S2$, while the $Data$ ones are used to send/receive data parameters, like $i2$ and $o2$, or to force the communication among the processes, like $sr1$.

- *proc* contains the processes. Among them, there are pools definition (S and D), tasks ($P12$, $P21$,...), intra and inter-communication buffers ($P3$, $P1$,...). Notably, at this stage the data objects are of type *node*, i.e. they do not reflect the privacy properties that will be associated later.

- *init* states the specification's initial process ($P3(eps)||P1(vnull)||$...) over which are applied the *hide*, *allow* and *comm* operators.

```
sort
PName = struct sssharing(Name)?is_sssharing|sscomputation(Name)?
      is_sscomputation;
Privacy = struct pair(frt:PName,snd:Nat);
Collection = Set(Data);
Data = struct node(value:Name)?is_node|eps|vnull|pnode(pvalue:Privacy)?
      is_pnode;
Name = struct satellite data|share1|share2|...;
map
empty: Data->Bool;
var
d2:Data;
eqn
(d2==eps) -> empty(d2) = true;
(d2!=eps) -> empty(d2) = false;
act
StartEvent_D,StartEvent_S,S1,S2,D1,...:Collection;
sr4,sr2,sr3,sr5,o2,i,o0,i1,o3,i2,o1,i0,... :Data;
proc
%Inter-communication buffer to send "share2" from S2 to D1
P3(d17:Data) = ((sum d18:Data.i2(d18).P3(d18))+(!empty(d17))
      ->(o2(d17).P3(eps)));
%Intra-communication buffer to send "share2" from S1 to S2
P1(d14:Data) = ((sum d15:Data.i0(d15).P1(d15)) + (o0(d14).P1(vnull)));
%Space Agency pool
S=((P11.P12.P13.t0.((tau.P14)+(tau.P15)).t0.P16.P17.P18)||(t0.P19.t0));
%Create share or S1 task
P12 = S1({node(share2),node(share1),node(satellite_data)}).
      o13(node(share1)).o1(node(share2));
%Compute satellite collision or S3 task
P19 = sum d12:Data.i13(d12).S3({d12,node(result1)}).o7(node(result1));
%Start event of the S party
P11 = StartEvent_S({});
%Send share or S2 task
P13 = sum d9:Data.i1(d9).S2({d9}).o3(d9);
%Data Center pool
D=(P20.P21.P22.P23.P24);
%Receive data or D1 task
P21 = sum d9:Data.i3(d9).D1({d9}).t18([d9]).o11(d9);
P20 = StartEvent_D({});
....
init hide ({sr4,sr2,sr3,sr5,...}, allow({sr4,sr2,sr3,sr5,S1,S2,D1,...},
    comm ({o2|i3->sr5,o0|i1->sr3,...},P3(eps)||P1(vnull)||S||D||...)));
```

Listing 3.1: *mCRL2* specification of the example in Figure 11.

3.3 Verification

Once the collaboration specification is generated, the user can choose the verification to execute on the model. In Section 3.1, we listed all the properties supported by our approach; here we will see in detail how the specification is enhanced to verify each of them.

3.3.1 Task Verification

This verification focuses on discovering if a task T knows the values of a chosen set of data objects existing in the model. To this aim, we define the following formula.

Definition 3.3.1 (Task Formula)**.** *Given a task T with a knowledge set of dimension m and a set of data $D = \{d_1, ..., d_n\}$, the property does T know about D? is expressed as:*

$$< true^*.exists\ e_1, ..., e_{m-n} : Data.T(\{e_1, ..., e_{m-n}, d_1, ..., d_n\}) > true$$

The formula $< f > true$ corresponds to the diamond modality, which is satisfied whenever there exists a path where the formula f is satisfied. $true^*$ means that any sequence of actions can be performed before T. $exists\ e_1, ..., e_{m-n}\ :\ Data$ defines placeholders for parameters of type $Data$, which are used to simulate the value of the other elements inside the knowledge of the action task T.

As an example of task verification, let us consider again our running example in Figure 11. It could be interesting to check if the task $S6$ gets to know $result2$, since it is essential for the reconstruction task. For this verification, the following formula is automatically generated:

$$< true^*.S6(\{node(result2)\}) > true$$

The verification's result is $true$, meaning that task $S6$ knows the data. One of the possible paths leading to the point in which the formula is satisfied is also automatically generated:
($StartEvent_D$,[]) $\rightarrow$ ($StartEvent_S$,[]) $\rightarrow$ ($S1$,[$share1, share2, satellite\ data$]) $\rightarrow$ ($S2$,[$share2$]), ($S3$,[$share1, result1$]) $\rightarrow$ ($S4$,[$atmospheric\ info, weather$])$\rightarrow$ ($D1$,[$share2$]) $\rightarrow$ ($D2$,[$share2, result2$]) $\rightarrow$ ($D3$,[$result2$]) $\rightarrow$ ($S6$,[$result2$]).

3.3.2 Participant Verification

This verification focus on checking if a participant P in the collaboration knows a selected set of data objects D. Instead of directly expressing this property as a μ-calculus formula, we define a new mCRL2 function that encodes this property inside the collabaoration specification.

Definition 3.3.2 (Contain function). *Given two parameters* $c1$ *and* $c2$ *of type* $Collection$*, the CONTAIN function returns* $true$ *if* $c2$ *is in* $c1$*, otherwise* $false$*.*

```
sort Collection = Set(Data);
map
CONTAIN : Collection # Collection → Bool;
var
c1,c2:Collection;
eqn
((c1*c2) == c2) -> CONTAIN(c1,c2) = true;
((c1*c2) != c2) -> CONTAIN(c1,c2) = false;
```

Listing 3.2: CONTAIN function.

In the listing above, the mCRL2 operator $*$ *stands for the intersection between sets. Therefore, the* $CONTAIN$ *function returns* $true$ *only if the intersection between* $c1$ *and* $c2$ *is equal to* $c2$*, so if every element of* $c2$ *is in* $c1$*.*

To exploit this function, we need to collect all the data objects that a participant produces and receives during its execution.

Definition 3.3.3 (Participant memory). *Given a party* P *in a collaboration, its knowledge (i.e. the set of elements of type* $Data$ *that are gained by the execution of its tasks) is stored in a process defined as follows:*

$$M(c : Collection) = sum\ c1 : Collection.\ t(c1).M(c + c1)$$

where t *is an action that will synchronise with other* t *actions added in the task processes of the same participant, and* $c1$*, i.e. the new collection of data received, will extend the current knowledge* c *of the memory. In particular, if* A_i *denotes the set of receiving processes* ($sum\ e_1 ... e_j : Data.i_1(e_1, ..., e_j). \ldots sum\ e_1 ... e_k : Data.i_n(e_1, ..., e_k).$) *and* A_o *denotes the set of sending actions* $(o_1(\ldots)\ldots o_m(\ldots))$ *as defined in Def. 3.2.6, the new specification for each task* $T \in P$*, with party memory* M *as in Def. 3.3.3, will be:*

$$A_i.n(c).t(c).A_o$$

where t *is forced to synchronise using the* $allow$ *and* $comm$ *operator as follows:*

$$allow(\{sr\}, comm(\{t|t \rightarrow sr\}, \ldots)$$

We can extend the above definition to make it check the $CONTAIN$ function in the following way.

Definition 3.3.4 (Memory in participant verification)**.**

$$M(c : Collection) = sum\ c1 : Collection.\ t(c1).(!CONTAIN(c + c1, D)) \\ \rightarrow M(c + c1) <> CONTAIN.delta$$

where D is the data objects set to be checked. If the function call $CONTAIN(c+c1, D)$ returns true, then the $CONTAIN$ action is executed, and the process stops; otherwise, the process starts again but with the collection updated with the new data objects.

At this point, the verification is reduced to a *reachability problem* of the $CONTAIN$ action. If the party knows the set of data D we also provide the trace leading to that state.

Given the example in Figure 2 we can check if party D gets to know $result1$ and $share1$, since this would lead to a case in which the secret sharing is violated. The specification will be updated accordingly.

```
proc
...
P26(d39:Collection) = sum d40:Collection.t18(d40).
    (!CONTAIN(d39+d40,{share1,result1}))->P26(d39+d40)<>CONTAIN.delta
...
```

Listing 3.3: *mCRL2* specification specialisation for participant verification.

In this example the result of the verification is $false$, meaning that the CONTAIN function is not satisfied and party D does not know about $share1$ and $result1$. Checking for the previous formula thus does not ensure that the secret sharing protocol is not violated. We will show how to check that property in the following subsection.

3.3.3 Secret Sharing Verification

This form of verification aims at checking if the secret-sharing technology implemented using the SSSharing, SSComputation, and SSReconstruction stereotypes is violated in the model or not.

Since the tasks associated with the mentioned PETs enhance the data objects with privacy features, e.g. the task with SSSharing stereotype produces a secret that must not be revealed, we need to reflect this characteristic on the data objects. To this aim we resort to the following definition that, in its own turn, makes use of the mapping function defined in Def. 3.2.4.

Definition 3.3.5 (Secret sharing data generation)**.** *Given a task with identifier* id *and the set* D *of data object names, composed by* D_i *and* D_o *that are respectively the set of input and output data object names, we define the function* $\mathcal{T}_{pet}$*, generating the PET-aware data object specifications, as follows:*

$$\mathcal{T}_{pet}(id, D) = \begin{cases} \{pnode(pair(sssharing(d)), n)) \mid d \in D_o\} \\ \qquad\qquad\qquad\qquad \textit{if } pet(id) = SSSharing \\ \\ \{pnode(pair(sscomputation(d), n)) \mid d \in D_o\} \\ \qquad\qquad\qquad\qquad \textit{if } pet(id) = SSComputation \end{cases}$$

where n *is a fresh natural number used to identify all the shares created from the task* id *in case it has the SSSharing stereotype, or identifies the group id coming from the task that has generated the object in case of SSComputation stereotype.*

Listing 3.4 shows how the function $\mathcal{T}_{pet}$ is used to generate PET-aware data objects on the example in Figure 11. It transforms $node$ objects into $pnode$ by associating the corresponding $Pname$ and id value.

From the example, we can notice that $share1$ and $share2$ have the same id, i.e. 2, since they belong to the same secret sharing task; it happens the same with $result1$ and $result2$, since they are generated from tasks in the same computation group.

```
proc
...
%secret share task
P12 = S1({pnode(pair(sssharing(share2),2)),
          pnode(pair(sssharing(share1),2)),node(satellite data)}).
          o13(pnode(pair(sssharing(share1),2))).
          o1(pnode(pair(sssharing(share2),2)));
%compute satellite collision task
P19 = sum d12:Data.i13(d12).S3({d12,pnode(pair(sscomputation(result1),0))})
      .o7(pnode(pair(sscomputation(result1),0)));
%help compute satellite collision task
P22 = sum d9:Data.i11(d9).D2({d9,pnode(pair(sscomputation(result2),0))})
```

```
        .t20({d9,pnode(pair(sscomputation(resul21),0))})
        .o9(pnode(pair(sscomputation(share2),0)));
...
```

Listing 3.4: Specification excerpt with secret sharing data generation.

At last, we define a new sort named *Dlist* and the function *union* that, given two objects of type *Dlist*, concatenates their elements without allowing any repetition (Listing 3.5). *Dlist* and *union* are essential to execute the following verification functions.

```
sort Dlist = List(Data);
map
union: Dlist # Dlist ->Dlist;
var
l0,l1 :Dlist;
eqn
(head(l1) in l0) -> union(l0ld1) = union(l0,tail(l1));
(!(head(l1) in l0)) -> union(l0,l1) = union(l0<|head(l1),tail(l1));
(l0== []) -> union(l0,l1) = l1;
(l1== []) -> union(l0,l1) = l0;
```

Listing 3.5: *Dlist* data type.

Secret sharing technology violation occurs when a participant has:

1. **n or more shares of the same secret**, i.e. output of the same *SSSharing* task. To this aim, we introduce in the specification the definition of the *sslist* function (Listing 3.6), which collects all the shares belonging to the specific *SSSharing* task (through the id) in a new list of *Data*.

```
map
sslist : Nat # Dlist # Dlist → Dlist;
var
l1,l2 : Dlist;
n : Nat;
eqn
%sslist function
is_pnode(head(l1)) && is_sssharing(frt(pvalue(head(l1))))
&& snd(pvalue(head(l1))) == n ->sslist(n,l1,l2)
= sslist(n,tail(l1),l2<|head(l1));

!is_pnode(head(l1)) || !is_sssharing(frt(pvalue(head(l1))))
|| snd(pvalue(head(l1)))!= id ->sslist(n,l1,l2)
= sslist(n,tail(l1),l2);

(l1 == []) -> sslist(n,l1,l2) = l2;
```

Listing 3.6: Function to collect secret shares.

2. **n or more outputs of the same computation group**, i.e. outputs of $SSComputation$ tasks with the same group identifier. Again, we need to insert in the specification a function definition (Listing 3.7). The difference between function $sslist$ and $sscomp$ is given by the data types they are checking: $SSSharing$ and $SSComputation$, respectively.

```
map
sscomp : Nat# Dlist # Dlist → Dlist;
var
l1,l2 : Dlist;
n : Nat;
eqn
%sscomp function
is_pnode(head(l1)) && is_sscomputation(frt(pvalue(head(l1))))
&& snd(pvalue(head(l1)))==id-> sscomp(n,l1,l2)
= sscomp(n,tail(l1),l2<|l1);

!is_pnode(head(l1)) || !is_sscomputation(frt(pvalue(head(l1))))
|| snd(pvalue(head(l1)))!= n -> sscomp(n,l1,l2)
= scomp(n,tail(l1),l2);

(l1 == []) -> sscomp(n,l1,l2) = l2;
```

Listing 3.7: Function to collect secret data computations.

We define in Listing 3.8 the function $sssharingviolation$, which returns $true$ if the size of list l is greater than a particular number th, otherwise $false$, where th is the threshold number that indicates the minimum amount of shares to reconstruct a secret, while l is a list of shares coming from the computation of $sslist$ or $sscomp$.

```
map
sssharingviolation : Nat # Dlist → Bool;
var
l : Dlist;
th : Nat;
eqn
(#l>=th) -> sssharingviolation(th,l) = true;
(#l<th) -> sssharingviolation(th,l) = false;
```

Listing 3.8: Function to detect secret sharing technology violation.

These conditions are checked only on those parties that observe the *no-creation* and *no-reconstruction* properties, which means no $SSSharing$ task with id number equal to the one currently checked nor $SSReconstruction$ task exist in the party under consideration.

Since the verification must be done on all the data objects passing through a specific party, we will enrich its $Memory$ process to check the violation property.

Definition 3.3.6 (Memory in stereotype verification). *Given a participant memory (as defined in Def. 3.3.3) it can be updated to check n violation functions in the following way:*

$$\begin{aligned} M(l : Dlist) = sum\ l_1 : Dlist.\ t(l_1).((\langle \textit{insert check}\rangle_1) &\rightarrow VLT(l_2).delta \\ <> ((\langle \textit{insert check}\rangle_2) &\rightarrow VLT(l_2').delta \\ <> ... \\ <> ((\langle \textit{insert check}\rangle_n) &\rightarrow VLT(l_2'').delta \\ <> M(union(l, l_1)) \end{aligned}$$

M is a process that collects objects of type $Dlist$, $\langle$insert check$_i\rangle$ is a placeholder for the i-th function to be checked, VLT is an action of type $Dlist$ used to express that a violation occurred, and l_2, l_2', l_2'' are the data objects involved in the violation.

If a party does not have a $SSReconstruction$ task (no-reconstruction property), it cannot know a number of computed shares greater than the threshold. We check that by enhancing the memory in stereotype verification (Def. 3.3.6) with the following function call.

$$\begin{aligned} &sssharingviolation(th, sscomp(n_1, union(l, l_1), [])) \\ &\qquad\qquad \rightarrow VLT(th, sscomp(n_1, union(l, l_1), [])) \\ &<> .... \\ &<> sssharingviolation(th, sscomp(n_n, union(l, l_1), [])) \\ &\qquad\qquad\qquad \rightarrow VLT(th, sscomp(n_n, union(l, l_1), [])) \end{aligned}$$

We call this function for every computed shares such that $n_1 \neq n_2 \neq \ldots \neq n_n$.

Moreover, the following checks are added in case a party does not have a $SSSharing$ task (no-creation property) or, if it has, the identifier n of the shares it generates is different from the checked ones, i.e. $n \neq n_1, \ldots, n \neq n_m$.

$$
\begin{aligned}
&sssharingviolation(th, sslist(n_1, union(l, l_1), []))\\
&\qquad\qquad \rightarrow VLT(th, sslist(n_1, union(l, l_1), []))\\
&<> \ldots.\\
&<> sssharingviolation(th, sslist(n_m, union(l, l_1), []))\\
&\qquad\qquad\qquad \rightarrow VLT(th, sslist(n_m, union(l, l_1), []))
\end{aligned}
$$

Listing 3.9 shows the enhancement of the memory process for our running example. Since the party D respects the no-creation and no-reconstruction properties, its memory will be updated to check the secret sharing violation function on $sslist$ and $sscomp$. This cannot happen for the party S, since it does not satisfy both properties.

```
proc
...
%Data center memory process
P31(d57:Dlist) = (sum d58:Dlist.t24(d58)
.(sssharingviolation(2,sscomp(0,union(d57,d58),[])))
->(VIOLATION(sscomp(0,union(d57,d58),[])).delta)
<>(sssharingviolation(2,sslist(2,union(d57,d58),[])))
->(VIOLATION(sslist(2,union(d57,d58),[])).delta)
<>P31(union(d57,d58)));
...
```

Listing 3.9: Embedding of secret sharing violation checking in the running example specification.

As last step, a formula verifying the existence of the VLT action is generated to execute the verification.

$$
< true * .exists \quad d : Dlist.VLT(d) > true \tag{3.1}
$$

In this case, the verification of the formula returns the result $false$, meaning that no violation occurs.

3.3.4 Additive Secret Sharing Verification

Additive secret sharing is a specialisation of secret sharing and differs from it just for the threshold value. Given a secret x dived in n shares, the threshold must be equal to n to reconstruct the secret. It is implemented using the stereotypes *AddSSsharing*, *AddSSComputation* and *AddSSReconstruction*.

Apart from the stereotypes used and the threshold value derived from the number of shares, the secret sharing, the additive secret sharing and the function secret sharing, as we will see later, share the same logic for detecting a violation.

This allows to use the same $\mathcal{T}_{pet}$ function for generating PET-aware data objects and, as consequence, we can apply the same violation functions as above (Listings 3.6, 3.7 and 3.8), and we can enhance the participant memory by following the same *no-creation* and *no-reconstruction* properties and, finally, we can verify the same VLT formula over the specification to get the verification result.

3.3.5 Function Secret Sharing Verification

Function secret sharing, like additive secret sharing, is another specialisation of secret sharing and is implemented by using the following stereotypes: *FunSSsharing*, *FunSScomputation* and *FunSSreconstruction*. This technology produces a fixed number of shares, equal to 2, which is also the threshold variable's constant value resulting in the specification.

The PETs mapping and verification enhancement follow the same rules for secret sharing and additive secret sharing violation verification.

3.3.6 Public Key Encryption Verification

This type of verification focus on checking that the public key encryption technology, implemented using the stereotypes *PKEncrypt*, *PKComputation* and *PKDecrypt*, is not violated in the model.

Tasks having the above PETs associated enhance data objects with a privacy feature that differs from the one defined for secret sharing tasks. Moreover, in models with public key encryption technology, tasks are not the only elements with associated stereotypes, data objects can have *PKPublic* or *PKPrivate* stereotypes associated.

Definition 3.3.7 (Encryption data generation). *Given a task with identifier id and the set D of data object names, composed by D_i and D_o that are respectively the set of input and output data object names, we define the function $\mathcal{T}_{pet}$, generating the PET-aware data object specifications for encryption verification,* *as follows:*

$$\mathcal{T}_{pet}(id, D) = \begin{cases} \{pnode(pair(cipher(d)), n) \mid d \in D_o\} \\ \qquad\qquad \textit{if } pet(id) = PKEncrypt \\ \qquad\qquad \textit{or } pet(id) = PKComputation \\ \\ \{pnode(pair(decodingkey(d)), n) \mid d \in D_i\} \\ \qquad\qquad \textit{if } pet(d) = PKPrivate \\ \qquad\qquad \textit{and } pet(id) = PKEncrypt \end{cases}$$

where n is a fresh natural number that identifies the key pair (public and private key) used to encrypt the data object.

The public key encryption technology is violated when a participant has:

- **the ciphertext** (the corresponding specification enhancement is in Listing 3.10);

```
map
haschipher : Dlist # Nat -> Data;
var
l : Dlist;
n : Nat;
eqn
%has cipher function
is_pnode(head(l)) && is_cipher(frt(pvalue(head(l))))
&& snd(pvalue(head(l))) == n -> haschipher(l,n) = head(l);

!(is_pnode(head(l))) || !(is_cipher(frt(pvalue(head(l)))))
|| snd(pvalue(head(l))) != n-> haschipher(l,n) = haschipher(tail(l),n);

(l == []) -> haschipher(l,n)=eps;
```

Listing 3.10: Function to detect the ciphertext with the given *id* value.

- **the decoding key** (the corresponding specification enhancement is in Listing 3.11).

```
map
haskey : Dlist # Nat -> Data;
var
l : Dlist;
n : Nat;
eqn
%has decoding key function
is_pnode(head(l)) && is_decondingkey(frt(pvalue(head(l))))
&& snd(pvalue(head(l))) == n-> haskey(l,n) = head(l);

```

```
!(is_pnode(head(l))) || !(is_decondingkey(frt(pvalue(head(l)))))
|| snd(pvalue(head(m))) != n -> haskey(l,n) = haskey(tail(l),n);

(l==[])->haskey(l,n)=eps;
```

Listing 3.11: Function to detect the decoding key with the given *id* value.

The *hascipher* function (Listing 3.10) verifies if a data parameter of type *cipher* with identifier n exists in the list of data parameters. If it does, that element is returned; otherwise, *eps*, i.e. the empty element, is returned. The *haskey* function (Listing 3.11) is similar, but it verifies if the data object is of type *decodingkey*.

The encryption is violated when a participant receives both data given by the *hascipher* and *haskey* functions (Listing 3.12).

```
map
encryptionviolation : Data # Data -> Bool;
var
d1,d2 :Data;
eqn
%encryption violation function
(d1!=eps && d2!=eps) ->encryptionviolation(d1,d2) = true;

(d1==eps || d2 == eps) -> encryptionviolation(d1,d2)=false;
```

Listing 3.12: Function to detect encryption violation.

These conditions are checked only on those participants that satisfy the *no-encryption* and *no-decryption* properties, which means that they are neither creating the ciphertext nor decrypting it. As for the secret sharing technology, we will enhance the memory of the chosen participant (Def. 3.3.6) with the following functions call:

$$encryptionviolation(haskey(union(l,l_1),n_1),hascipher(union(l,l_1),n_1))$$
$$\rightarrow VLT([haskey(union(l,l_1),n_1)]++[hascipher(union(l,l1),n_1)])$$
$$<> ....$$
$$<> encryptionviolation(haskey(union(l,l_1),n_m),hascipher(union(l,l_1),n_m))$$
$$\rightarrow VLT([haskey(union(l,l_1),n_m)])++[hascipher(union(l,l1),n_m)])$$

The *encriptionviolation* function is called for each pair of *haskey* and *hascipher* functions computed over the same identifier n. If the result is

$true$, the VLT action is executed, since a violation exists. Otherwise, it continues with the subsequent checks until the process starts again and receives new data objects.

3.3.7 Symmetric Key Encryption Verification

Symmetric key encryption is another specialisation for encryption. Three stereotypes implement it: *SKEncrypt*, *SKComputation* and *SKDecrypt*.

In this case, there is no pair of keys used to encrypt/decrypt, since the key used to encrypt the data object is the same used to decrypt it. We depict this feature on the data object as follows.

Definition 3.3.8 (Symmetric data generation)**.** *Given a task with identifier id and the set D of data object names, composed by D_i and D_o that are respectively the set of input and output data object names, we define the function $\mathcal{T}_{pet}$, generating the PET-aware data object specifications for symmetric key encryption, as follows:*

$$\mathcal{T}_{pet}(id, D) = \begin{cases} \{pnode(pair(cipher(d)), n) \mid d \in D_o\} & \\ \qquad\qquad \textit{if } pet(id) = SKEncrypt & \\ \qquad\qquad \textit{or } pet(id) = SKComputation & \\ & \\ \{pnode(pair(decodingkey(d)), n) \mid d \in D_i\} & \\ \qquad\qquad \textit{if } pet(d) = Key & \\ \qquad\qquad \textit{and } pet(id) = SKEncrypt & \end{cases}$$

where n is a fresh natural number that identifies the key used to encrypt the data object.

As for additive and function secret sharing, public key encryption and symmetric key encryption also share the same logic for the violation verification. Then we can reuse the functions in Listings 3.10, 3.11 and 3.12.

3.3.8 Reconstruction Verification

This property is related to the correct design of the secret sharing protocols. The technology implementation is correct if it is always possible to reconstruct the secret. Otherwise, there is an error.

The mapping of the data objects follows the one already defined in Def. 3.3.5, while the function to identify the violation is different (Listing 3.13).

```
map
list2bag : Dlist # Bag(Nat) -> Bag(Nat);
is_rencostructed : Bag(Nat) # Nat -> Bool;
var
l : Dlist;
b: Bag(Nat);
th: Nat;
eqn
%List of data to bag of shares and computed shares function
is_pnode(head(l)) && (is_sssharing(frt(pvalue(head(l))))
||is_sscomputation(frt(pvalue(head(l))))) -> list2bag(l,b)
= list2bag(tail(l),b+{snd(pvalue(head(l))):1});

!is_pnode(head(l)) || !is_sssharing(frt(pvalue(head(l))))
|| !is_sscomputation(frt(pvalue(head(l)))) -> list2bag(l,b)
= list2bag(tail(l),b);

(l == []) -> list2bag(l,b) = b;
%reconstruction function
(exists n:Nat. n in b && count(n,b) >= th)
-> is_rencostructed(b,th) = true;

(!(exists n:Nat. n in b && count(n,b) >= th))
-> is_rencostructed(b,th) = false;
```

Listing 3.13: Functions for reconstruction checking.

The $list2bag$ function (Listing 3.13) is gathering together the shares and computation shares with the same identifier, taking into account their multiplicity. Then, $is\_reconstructed$ checks the multiplicity: if it is greater than the threshold it returns $true$, otherwise $false$. This function has to be checked when the reconstruction is taking place, i.e. when the task marked with the *SSReconstruction*, *AddSSReconstruction* or *FunSSReconstruction* stereotype is performed.

Definition 3.3.9 (Reconstruction task enhanced with reconstruction verification). *Given a task process with identifier id, a set of input processes I, a knowledge base K and a set of output actions O, such that $pet(id) =$* SSReconstruction, *the task can be enhanced to check reconstruction as follows:*

$$I.id(K).O.(\,is\_reconstructed(\,list2bag(K,\{0:0\}),th))$$
$$\rightarrow NOVLT.delta <> VLT.delta$$

where $\{0:0\}$ initialises the bag containers.

If there exists a path that leads to the VLT action, then there is a trace in which the reconstruction is not possible. A PE-BPMN model without a reconstruction task does not need to compute any of these functions to return that a violation occurs.

Listing 3.14 shows how the task with the $SSReconstruction$ stereotype in Figure 11 is enhanced to execute the verification.

```
proc
...
P17 = sum d4:Data.i7(d4).sum d8:Data.i15(d8).S7({d4,d8,reconstructed}).
      (is_reconstructed(list2bag([d4,d8,reconstructed],{0:0}),2))
      ->(NOVLT.delta)<>(VLT().delta);
...
```

Listing 3.14: Task specification of the running example enhanced for reconstruction verification.

The formula (3.1) is used again to execute the verification. In the example, the secret is always reconstructed, as no VLT action is found.

3.3.9 MPC Verification

Similarly to the reconstruction verification, the MPC verification is related to checking the correct PETs usage. Tasks with the MPC stereotype and the same group id must be executed *synchronously*. If there is an execution in which one cannot start until the other one ends, or just one of them is executed, there is an error in the implementation of the protocol.

In this case, the focus is not on the data objects and their privacy features. Hence, it is not necessary the $\mathcal{T}_{pet}$ function to associate privacy features to data objects. Instead, we need to modify the MPC processes' definition to force synchronisation among the tasks.

Definition 3.3.10 (MPC tasks)**.** *Given a task with identifier id such that $pet(id) = MPC$, and let m be the number of tasks with its same group identifier n. We can enhance their process task specifications as follows:*

$$allow(\{\, sr \,\}, comm(\{\, s_1| \ldots |s_m \ \rightarrow sr\},$$
$$T_1 = I.s_1.id_1(k).O \ldots T_m = I.s_m.id_m(k).O))$$

where I is the set of input processes, O the set of output actions, and s is a fresh action added to all the tasks with the same MPC identification number.

The idea is to force the communication among the MPC tasks. Suppose the synchronisation cannot take place, then the specification will be deadlocked. If a deadlock exists in the model, the MPC synchronisation property is violated. Using the $mCRL2$ toolset, we can also derive the traces that lead to the deadlock. If the model that we are analysing already has a deadlock point, then we have a false positive, but we can avoid that by using the verification that we present below.

3.3.10 Deadlock Freedom

Deadlock freedom has always been an important property to be checked on systems. For BPMN collaboration models, being free from deadlock roughly means that every participant in the collaboration can terminate its execution. Instead, the $mCRL2$ toolset has a different way to define the deadlock of a specification, which corresponds to the property of never-ending. In $mCRL2$, deadlocks are states with no outgoing transitions [83].

Since $mCRL2$ deals with infinite-state model-checking [27] we need a workaround. Using the $mCRL2$ tool for detecting deadlock (i.e., the option "-D"), we retrieve all the traces that are considered deadlocked, and then we traverse every path to check if it contains the end events of all participants in the collaboration. If a deadlock with the BPMN meaning is detected, we can also return a counterexample to show where the deadlock is occurring.

By applying this verification to our running example, we obtain a list of traces, all of which containing both the end actions, meaning that no deadlock exists in the PE-BPMN model.

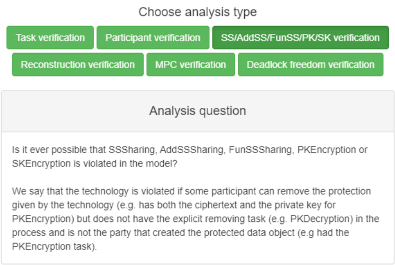
3.4 Tool Implementation

This section presents how the approach illustrated in the previous sections is supported in practice. The tool is an open-source software that uses the Java library jBPT [123] to generate the process trees in the control-flow transformation step. It can be redistributed and eventually modified under the terms of the GPL2 License. The source code as well as the user guide are publicly available, and they can be retrieved on-line[3].

The tool can be executed locally via a Java stand-alone application we made available[4]. It needs $mCRL2$ installed on the machine to carry out the verification. The application allows the user to load a PE-BPMN model to be verified in the *.bpmn* format. The graphical interface permits to select the verification and reports the verification results in a textual format. At the end of each analysis it is possible to find all the files generated by the tool in the "result_FSAT" folder, allowing the user to possibly exploit other tools provided by $mCRL2$.

The tool has been also integrated in Pleak[5] [47] as a plug-in. Pleak is a modelling and analysis environment for privacy-enhanced systems. It gives the possibility to model privacy enhanced systems and provides privacy audit features, like sensitive or guessing advantage analysis.

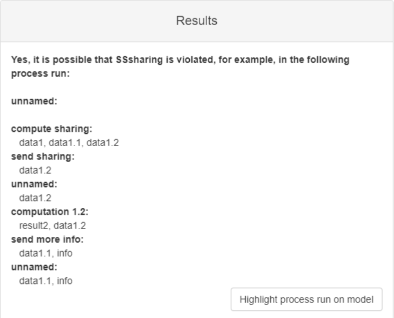
Figure 17 and 18 shows a screenshot taken from Pleak showing the view that opens while selecting the PET detection button in the PE-BPMN editor. The green buttons allow the user to select the properties he/she wants to verify, as discussed in Section 3.1. Once a selection is done, the



Is it ever possible that SSSharing, AddSSSharing, FunSSSharing, PKEncryption or SKEncryption is violated in the model?

We say that the technology is violated if some participant can remove the protection given by the technology (e.g. has both the ciphertext and the private key for PKEncryption) but does not have the explicit removing task (e.g. PKDecryption) in the process and is not the party that created the protected data object (e.g had the PKEncryption task).

Figure 17: Screenshot Pleak's verification selection



```
Yes, it is possible that SSsharing is violated, for example, in the following
process run:

unnamed:

compute sharing:
  data1, data1.1, data1.2
send sharing:
  data1.2
unnamed:
  data1.2
computation 1.2:
  result2, data1.2
send more info:
  data1.1, info
unnamed:
  data1.1, info
```

Figure 18: Screenshot Pleak's verification result

[3] https://github.com/SaraBellucciniIMT/leakDetectionAnalyzer

[4] https://github.com/SaraBellucciniIMT/leakDetectionAnalyzer/blob/master/leakdetect_v3.1(22012021).jar

[5] https://pleak.io/home/

"analysis question" box shows a description of the property to support the understanding of the results. The "Results" box contains the verification results. In case of a property violation, the tool provides the path leading to the corresponding state.

In case of a violation, to further support the user in solving the issue, the "Highlight process run on model" button gives the possibility to show the corresponding counterexample, which will be displayed directly on the model by colouring the tasks and data objects involved in the violation (see the grey and red-coloured elements in Figure 19).

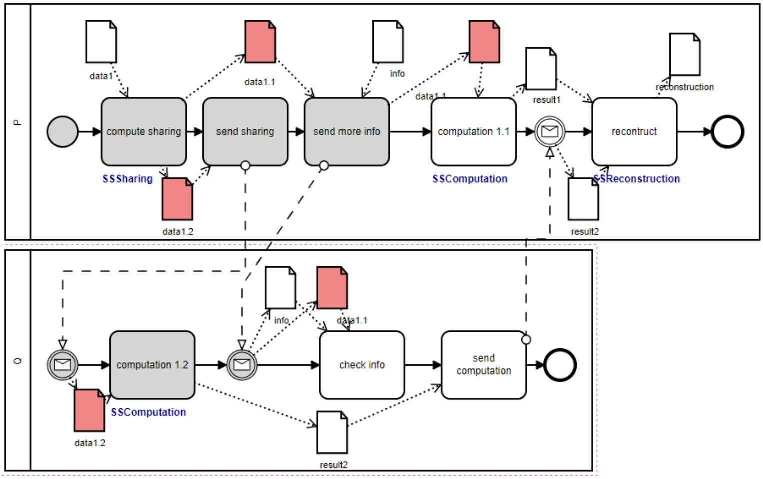


Figure 19: PE-BPMN where the tasks and data involved in the violation of Figure 18 are highlighted.

3.5 Validation

In this section, we discuss the validation we performed to assess the correctness, performance, and scalability of the tool.

By *correctness* we mean the ability to obtain a specification that reproduces the behaviour of the PE-BPMN model in input, ensuring that the results obtained for the specification are valid also for the model. By *performance* we mean the ability to compute the specification and the verification results in a reasonable time, allowing to gain real benefit from their application. By *scalability* we mean the ability of the tool to handle the verification of PE-BPMN models of increasing size. This validation shows the potentiality but also the limitation of the approach.

The evaluation we have performed has been then structured over two main parts. The first one refers to pseudo-real models inspired by possible real case studies and corner cases to test the most likely cases. The models are available on GitHub[6]. The reason for using existing repositories of PE-BPMN models is to assess the feasibility of the tool, showing that it can also process real(istic) models designed by third-parties, without knowing a priori the quality of the designed models.

The second one extensively validated the proposed verification approach by running a scalability analysis via ad-hoc designed and synthetically generated models. The rationale for using synthetic models was to do a scalability analysis on models on which we have complete control and show the tool capabilities in extreme scenarios.

The parameters that we take into account in both evaluations are: the *number of pools* (#pool), the *numbers of elements represented as processes* like tasks and intermediate message events (#task), the *number of data exchanges* among the elements, either in the same participant or not (#data exc.), the *time needed to transform the collaboration into a $mCRL2$ specification* (T. time) and the *time to execute the verification* over it (V. time), the time measure is in milliseconds.

#pool, *#task* and *#data exc.* suggest the size of the model and, consequently, the size of the specification generated from it. *T. time* measures the performance of our methodology that consists of generating the specification. While the *V. time* parameter actually depends on the $mCRL2$ toolset, but we need to measure it because the verification part makes this methodology important and reveals the quality of the specification.

All the experiments have been performed on a dedicated machine running Windows 10 Pro 64 bits, equipped with a processor Intel(R)

[6] `https://github.com/SaraBelluccini IMT/leakDetectionAnalyzer/tree/master/pe-bpmn%20models`

Core(TM) i7-5500U CPU, and 8 GB of RAM (but only 256MB allocated to the Java heap).

3.5.1 Experiments on Realistic PE-BPMN Models

Tables 2, 3, 4, 5 and 6 show a first experiment performed on a set of pseudo-real PE-BPMN models modelled with the Pleak tool. Since every model is unique, and it is not easy to correlate them, we postpone the scalability validation to the next Section 3.5.2 while focusing on the correctness of the approach and glimpsing some hints about the performance evaluation.

Model	#pool	#task	#data exc.	T. time (ms)	V. time (ms)
Model10	2	15	12	206	666
Model11	2	13	8	89	755
Model12	2	8	3	12	262
Model13	2	8	3	12	264
Model14	3	30	28	103	40762
Model15	3	18	12	20	566
Model16	2	19	16	18	3833
Model17	1	4	1	4	268
Model26	2	9	6	22	319
Model27	2	9	6	23	342
Model6	2	14	10	20	518
Model8	2	17	14	14	1409
Model9	4	26	18	53	12473
Model1	2	14	11	8	500
Model2	2	14	9	15	1021
Model3	3	30	28	26	168357
Model4	2	14	9	11	651
Model5	2	7	3	4	344
Model7	4	31	22	63	153849

Table 2: Results of secret sharing and its specialisations, additive and function, verification over PE-BPMN models. The rows in red are models with a violation.

For every table, we use the red row to highlight when there is a **vio-**

lation, which means that secret sharing (Table 2) or encryption (Table 3) are violated, the reconstruction of the secret is not always possible (Table 4), there exists a case in which the synchronisation among MPC task does not happen (Table 5), or there exists a deadlock in the model (Table 6). The reconstruction property is checked over the same models of the secret-sharing verification, while the deadlock property applies to the overall sets of models used in precedence.

Model	#pool	#task	#data exc.	T. time (ms)	V. time (ms)
Model22	2	18	12	188	5557
Model23	4	16	9	26	1775
Model24	2	7	3	11	412
Model25	2	8	3	20	366
Model34	3	27	9	99	58252
Model18	2	8	3	16	526
Model19	2	8	3	7	428
Model20	2	8	3	6	652
Model21	2	8	3	9	393
Model33	3	27	9	25	10726

Table 3: Results of public key and symmetric key encryption verification for PE-BPMN models. The rows in red are models with a violation.

In terms of *correctness*, we obtained the expected results for all the verification types, since the correct results are known a priori. Regarding *performance*, all the conducted experiments demonstrate that the translation time remains unaffected despite an increase in the model's dimension. On the contrary, we can notice that in Table 2 even if Model14 and Model7 have similar characteristics, the verification time of the model with violation is longer than the safe one. In contrast, in Table 3 with Model34 and Model33, we obtain an opposite trend.

This contrasting result can probably be explained by the generation of the counterexample path and the number of data exchanges executed by the models under analysis. Since the communications (or data exchange) greatly contributes to the growth of the state space and the computation of the counterexample needs to inspect it, we notice this behaviour that

Model	T. time (ms)	V. time (ms)	Model	T. time (ms)	V. time (ms)
Model10	221	636	Model6	12	497
Model11	112	0	Model8	13	462
Model12	21	460	Model9	30	10117
Model13	10	0	Model1	10	463
Model14	74	1734	Model2	15	0
Model15	18	1018	Model3	35	0
Model16	16	643	Model4	17	0
Model17	4	449	Model5	5	0
Model26	19	682	Model7	34	17498
Model27	17	2848			

Table 4: Results of reconstruction verification over the same PE-BPMN models of Table 2. The rows in red are models that contain a trace in which the secret is not reconstructed.

Model	#pool	#task	#data exc.	T. time (ms)	V. time (ms)
Model28	2	8	3	165	308
Model29	2	7	0	27	582
Model30	2	8	0	17	432
Model31	2	12	0	37	671
Model32	2	12	4	30	440

Table 5: Results of MPC verification over PE-BPMN models. The rows in red are the ones in which a path exists such that the synchronisation is not satisfied.

will be confirmed in the validation over synthesised models.

3.5.2 Experiment on Synthesised Models

To measure the *performance* and *scalability* of the approach, we decided to apply our methodology on two models, one containing a violation of the technology and one that preserves it, in which we grow linearly the number of pools it contains, i.e. increasing the parallelism inside the specification, which is the most critical point for techniques based on a process algebra.

Model	T. time (ms)	V. time (ms)	Model	T. time (ms)	V. time (ms)
Model22	274	525	Model12	6	385
Model23	30	372	Model13	6	608
Model24	10	520	Model14	16	1624
Model25	10	404	Model15	12	427
Model34	81	3741	Model16	13	691
Model18	13	387	Model17	5	451
Model19	7	394	Model26	12	1478
Model20	7	694	Model27	15	552
Model21	7	359	Model6	8	922
Model33	28	4214	Model8	10	578
Model28	7	383	Model9	36	2186
Model29	8	340	Model1	8	672
Model30	5	522	Model2	43	1158
Model31	14	528	Model3	16	1957
Model32	12	440	Model4	9	711
Model10	11	421	Model5	4	560
Model11	13	916	Model7	30	715

Table 6: Result of deadlock freedom verification. The red rows are the ones with a deadlock.

Even if the verification time is not entirely dependent on our contributions, since we use tools available in the $mCRL2$ toolset, we decided to make it part of the validations because it is an essential point highlighting the importance of the methodology. Moreover, another metric that has been considered is the Extended Cyclomatic Metric which measures the complexity of the model's behaviour [76]. The measurement is done using the RePROSitory tool [38].

Figure 20 shows the model with the violation, which is $pool2$ participant receives both secret shares $data1.2$ and $data1.1$ and is then able to reconstruct the secret even if it has no right to do so.

As we can notice the violation point has been inserted after the last communication ("Check info" task) to consider the worst-case scenario. To increase the parallelism, we add at each verification round a new pool

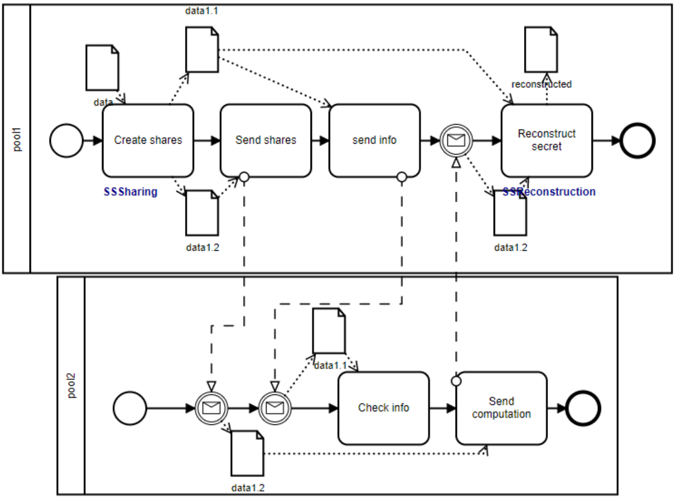


Figure 20: PE-BPMN model with violation used to measure the scalability of the approach.

P. The process of P is the sequence of an intermediate message event and a task, and operates as a medium to send data1.2. P will receive data1.2 from pool1 (new message flow between "Send share" and the message event in P), and then P will send it to pool2 (message flow between the task in P and the message event in pool2). Every new pool will increase the number of parties that data1.2 must traverse before going from pool1 to pool2.

The model used in the non-violation scenario is the same as in Figure 20 but, instead of sending the share named data1.1, a data object without privacy features is sent. We use the same technique as explained for the model with a violation to increase the number of pools.

As we can see from the results in Table 7, both for models with and without violation the verification time increases quadratically with respect to the number of pools. For example, (#pool=15, verif. time(min) = 16), (#pool=16, verif. time(min) = 38) and (#pool=17, verif. time(min) = 88); this trend is also highlighted in the chart of Figure 21. On the other hand, the translation time seems to increase more or less linearly with the complexity of the model, as shown in Figure 22.

Comparing the analysis on the two types of models, we can see that it is always the case that the one with a violation takes more time to get an answer with respect to the same model without it, confirming the fact that the counterexample generation affects the verification time when the property is not satisfied. We fixed a threshold of 2 hours for the verification time. The results shows that our solution achieves a quite good scalability, as we can analyse 17 pools in parallel.

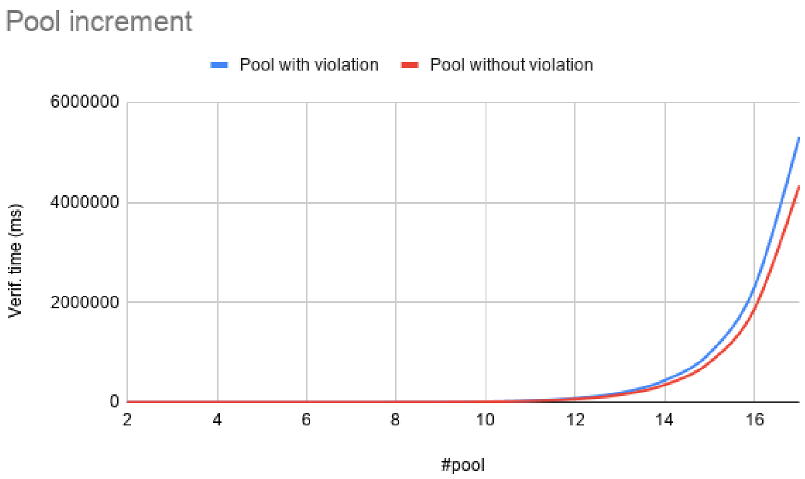


Figure 21: Chart showing how the verification time for PE-BPMN models in table 7 grows depending on the number of pools being in parallel.

3.6 Related Work

In this section, we discuss the most relevant attempts in formalising BPMN models without and with data, and we compare our work with other verification approaches.

On Formalising BPMN. Several formalisations have been proposed in order to disambiguate the semi-formal semantics of BPMN. The most

Model	#pool	#task	#data exc.	Extended cyclomatic	T. time (ms)	V. time (ms)
MODELS WITH VIOLATION						
pool_vlt1	2	13	9	11	3	322.5
pool_vlt2	3	17	11	14	5.5	404.5
pool_vlt3	4	21	13	17	7	522.5
pool_vlt4	5	25	15	20	6	604.5
pool_vlt5	6	29	17	23	9	725
pool_vlt6	7	33	19	26	7	1269
pool_vlt7	8	37	21	29	11.5	2627
pool_vlt8	9	41	23	32	13	5921.5
pool_vlt9	10	45	25	35	8.5	13920
pool_vlt10	11	49	27	38	15	33290
pool_vlt11	12	53	29	41	12	80468
pool_vlt12	13	57	31	44	14	188920.5
pool_vlt13	14	61	33	47	17.5	440131.5
pool_vlt14	15	65	35	50	23	983674
pool_vlt15	16	69	37	53	21	2299189.5
pool_vlt16	17	73	39	56	26	5307371
MODELS WITHOUT VIOLATION						
pool_1	2	13	8	11	4	250.5
pool_2	3	17	10	14	7.5	596
pool_3	4	21	12	17	6	408
pool_4	5	25	14	20	7	438.5
pool_5	6	29	16	23	10.5	600
pool_6	7	33	18	26	8	1073
pool_7	8	37	20	29	9	2176
pool_8	9	41	22	32	11	4918.5
pool_9	10	45	24	35	10	11631
pool_10	11	49	26	38	11	27149
pool_11	12	53	28	41	14	66339
pool_12	13	57	30	44	13	149742
pool_13	14	61	32	47	14	351630
pool_14	15	65	34	50	16	799768
pool_15	16	69	36	53	17	1863411
pool_16	17	73	38	56	18	4331327

Table 7: Experiments over a set of synthesised models in which the number of pools grows linearly. The starting model for *"pool with violation"* is in Figure 20, while for *"pool without violation"*, we constructed the same model where, instead of "data1.1", that is the share triggering the violation, a data that is not a share is sent.

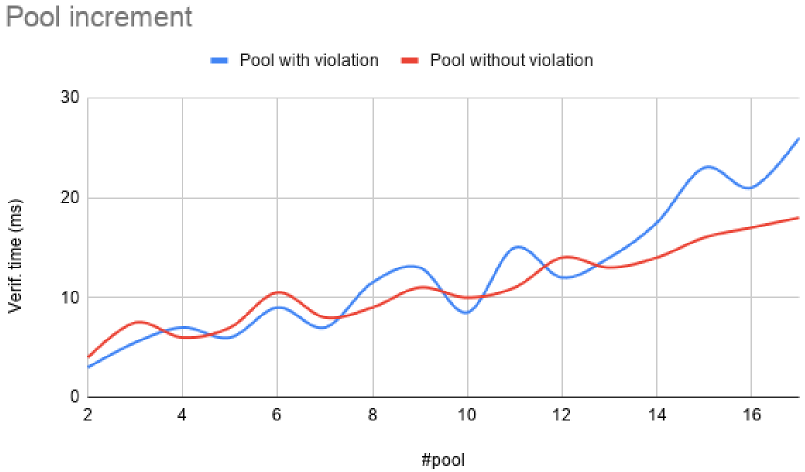


Figure 22: Chart showing how the translation time for PE-BPMN models in Table 7 grows depending on the number of pools being in parallel.

common formalisations of BPMN are given via mappings to various formalisms focusing on core elements of the notation, such as Petri Nets [**huai˙towards˙2010**, 44, 71, 106, 15], and process calculi [103, 138, 102, 36, 99]. Some approach also translate processes into a model checker input language, e.g. [82] verify BPMN by translating it (via a Petri Net intermediate model) into the model checker input language TLA+. Others formalise BPMN directly into First-Order Logic [63] and then rely on a TLA+ implementation to carry out the formal verification. All these translation works abstract from data objects since TLA+ cannot deal with data and, consequently, are unsuitable for verifying security flows.

Considering process algebras, in [103] a translation to COWS is proposed in order to reason about qualitative and quantitative behaviour of the business process. However, the support for specifying and handling data is missing in the verification method. In [138] a formalisation from BPMN to CSP is proposed, and also in this case data objects are not considered and the refinement ordering used as verification method makes it difficult to construct behavioural properties like the one for verifying a sssharing violation. This kind of formalisations are influenced by the constructs of the used language and the features of the related verification techniques. None of these approaches supports the management of data, which represents a barrier on the verification of data related properties.

Focusing on BPMN with data, only few formalisations are available in the literature (e.g., [24, 48]). In [24] the authors propose a semantic framework for BPMN with data. This approach is based on BPMN 1.0 and has a one-process view, while our focus is on the communication among multiple processes, as we are interested in exchange of data including secrets among multiple collaboration parties. In [48], instead, BPMN models with data objects are formalised in terms of rewriting logic. Also in this case, collaboration scenarios are not considered, while they are of main importance in our approach. A recent work [37] provides a formal semantics of multi-instance collaborations taking into account the interplay between control features, messages and data. The formalisation is based on a BNF syntax of BPMN. This formalisation has driven the implementation of an animator tool that provides the visualization of the execution of the given model. However, a verification method is missing.

On Verification for Leakage Detection. Much effort has been devoted to the formalisation and verification of business processes (e.g., [89, 54, 50, 119]). Nevertheless, less attention has been paid to the security perspective over data of the models. Considering privacy issues and data leakage detection, some attempts have already been made using Petri Nets [5], process graphs [116] and also session types [30] to detect if a leakage exists and where. Unfortunately, these approaches are coarse-grained verification techniques that do not consider more advanced features, like the notion of data, instead of tokens, and they do not take into account and make difficult to represent security policies, like PETs.

In [6] the authors focus on solving the problem of data privacy by implementing, at design time, GDPR (General Data Protection Regulation) patterns without introducing new BPMN elements, but neither a way to apply verification nor validation is proposed. Regarding GDPR, in [16] the authors propose a systematic approach to operationalise it; in this respect, our proposal could be used in the last step to automatise the way of evaluating the solution, if PETs are used. In [111], an extension of BPMN with security policies expressed as queries is proposed, together with a way to analyse them. In this case, however, the user should learn two languages: the one for modelling, using the new elements, and the one to apply verification, to manually write the queries. In addition, the framework is not able to give a counterexample of a violation, which is an important hint to correct errors occurring at design time as fast as possible.

Moreover, the state of the art [25, 53, 7, 49, 29] highlights an increasing interest in enhancing business processes with privacy technologies, which consequently calls for methodologies to verify them.

Chapter 4

From Collaboration Logs to Formal Specifications

This chapter presents the *bottom-up approach* that bridges the gap between FM and BPM communities by taking inspiration from the process mining field for automatically generating formal models representing the behaviour of existing systems by analysing observations that the systems produced, i.e. logs, thus enabling formal verification.

Most of the approaches available in the BPM literature consider only the point of view of a single organisation. They do not provide techniques to derive a specification of a distributed scenario compositionally. On the other hand, FM usually defines its models manually, making it challenging to apply FM techniques in the business environment.

To overcome these issues, we rely on techniques from the process algebra community to exploit their inherent compositionality and comprehensive analytical tools and on process mining to generate models automatically.

The **PALM Methodology - (Process ALgebraic Mining)**, and its related software tool, aims at obtaining process algebraic specifications from system logs via a mining algorithm. The main phases of the methodology are shown in Figure 23.

The starting point is given by logs taken from components of real

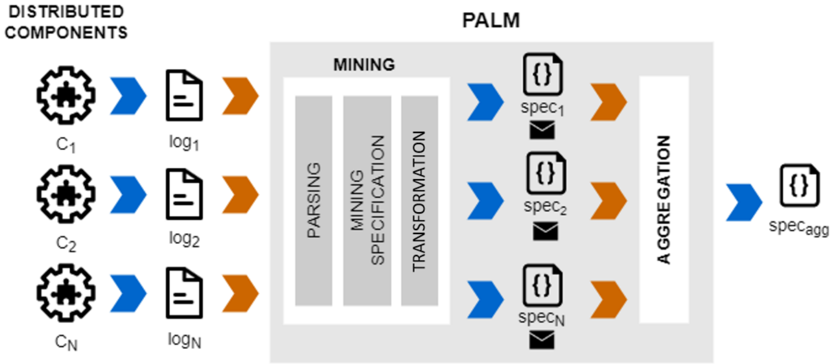


Figure 23: Overview of the PALM methodology.

world systems; those can be, e.g., logs of an industrial process as well as of a client-server network. In the *mining* step logs are analysed to generate a formal specification for each of them, together with a mapping associating sending/receiving action and with the exchanged messages.

The individual specifications can be exploited for verifying properties of the individual systems, e.g., by means of model checking techniques. But, more importantly, if the logs originate from components of a distributed system, the individual specifications can be combined, in the *aggregation* step, to obtain a formal model of the global system, which again can be analysed to consider issues originated by erroneous or unexpected interactions among the components.

The PALM methodology is implemented as a software tool that inputs one or more event logs and outputs the specification for each log. Additionally, in case of multiple logs belonging to a distributed system, PALM also outputs the global specification of the system. The inputs are logs expressed in the standard XES format, while the output is a customised $mCRL2$ specification.

Experiments have validated the methodology with both custom-made and real event logs.

The rest of the chapter provides a formalisation of the specification language used to model the output of the methodology (Section 4.1), presents the PALM methodology and the developed tool (Section 4.2), describes how the methodology and related tool can be used (Section 4.3) and reports on the empirical validation of the approach (Section 4.4). Finally a reviews of related works is given (Section 4.5). All the proofs related to the formal propositions can be found in Appendix A.

4.1 The $nCRL2$ Core Calculus

PALM uses a fragment of the $mCRL2$ specification language. We formalise in this section such language, which we call $nCRL2$ (nano $mCRL2$), and we show that the nano calculus respects the original semantics of $mCRL2$, enabling us to use the toolset available for $mCRL2$ on the specifications we generate with our methodology.
In Appendix A we report part of the semantics of $mCRL2$, and we provide the proofs of the propositions stated in this section.

In Def. 4.1.1 we formalise the syntax of $nCRL2$, where we use the following countable sets:

- The set $\mathbb{A}$ of *basic actions* (ranged over by a)
- The set $\mathbb{P}$ of *process variables* (ranged over by P)

Definition 4.1.1 ($nCRL2$ Process Specification Syntax)**.** *A process specification in $nCRL2$ is a pair $\langle p, E\rangle$ where E is a set of process equations of the form $P = p$ and p is a process expression defined by the following grammar*

$$\begin{array}{lll}
p ::= & & \textit{(Process expression)} \\
 & \tau & \textit{(Silent action)} \\
| & a & \textit{(Basic action)} \\
| & p + p & \textit{(Choice operator)} \\
| & p.p & \textit{(Sequence operator)} \\
| & p||p & \textit{(Parallel Composition operator)} \\
| & allow(V, p) & \textit{(Allow operator)} \\
| & comm(C, p) & \textit{(Communication operator)} \\
| & hide(I, p) & \textit{(Hiding operator)} \\
| & P & \textit{(Process equation call)}
\end{array}$$

where $V, I \subseteq \mathbb{A}$ and C is a set of allowed communications of the form $a_1|\ldots|a_n \to c$ where $n > 1$ and $a_1, \ldots, a_n, c \in \mathbb{A}$. For each P in the process specification, there exists a unique definition $P = p \in E$. We identify process expressions up to commutativity and associativity of choice and parallel compositions.

A process specification $\langle p, E\rangle$ can be written in the following format suitable as input for the $mCRL2$ toolset:

```
act
a_1 , a_2 , ... , a_n ;
proc
P_1 = p_1 ;... ; P_m= p_m ;
init p;
```

where $\{a_1, \ldots, a_n\}$ is the set of actions in $\langle p, E\rangle$, and $\{P_1 = p_1, \ldots, P_m = p_m\} = E$.

Definition 4.1.2 ($nCRL2$ Process Specification Semantics)**.** *The semantics of a process specification $\langle p, E\rangle$ is a LTS $(S, L, \to)$ as follow:*

- *The set of states S (ranged over by s) contains process specifications and one special termination state, denoted by $\checkmark$.*

- *The set of labels L (ranged over by α) is generated by the following grammar:*

$$\alpha ::= \tau \ \mid \ a \ \mid \ \alpha_1|\alpha_2$$

- *The transitions relation $\to \subseteq S \times L \times S$ is inductively defined by the operational rules in Table 8 Notice that we do not need symmetric versions of rules CH1, CH2, PAR1, PAR2 and PARC2 because, as already mentioned in Def. 4.1.1, we identify process expressions up to commutativity and associativity of choice and parallel operators. We will write $s_1 \xrightarrow{\alpha} s_2$ to indicate that $(s_1, \alpha, s_2) \in\to$ and, to improve the readability of the operational rules, we omit the set of process equations when they play no role in a rule, e.g. writing $\langle p, E\rangle \xrightarrow{\alpha} \langle p', E\rangle$ as $p \xrightarrow{\alpha} p'$, or $\langle p, E\rangle \xrightarrow{\alpha} \checkmark$ as $p \xrightarrow{\alpha} \checkmark$*

Where the communication function (γ_C) used by the $comm$ operator and the hiding function (θ_I) used by the $hide$ operator are defined as follows.

ACT^n	$\frac{}{\alpha\xrightarrow{\alpha}\checkmark}$	CH_1^n	$\frac{p\xrightarrow{\alpha}\checkmark}{p+q\xrightarrow{\alpha}\checkmark}$
CH_2^n	$\frac{p\xrightarrow{\alpha}p'}{p+q\xrightarrow{\alpha}p'}$	SQ_1^n	$\frac{p\xrightarrow{\alpha}\checkmark}{p.q\xrightarrow{\alpha}q}$
SQ_2^n	$\frac{p\xrightarrow{\alpha}p'}{p.q\xrightarrow{\alpha}p'.q}$	REC_1^n	$\frac{p\xrightarrow{\alpha}\checkmark}{\langle P,E\cup\{P=p\}\rangle\xrightarrow{\alpha}\checkmark}$
REC_2^n	$\frac{p\xrightarrow{\alpha}p'}{\langle P,E\cup\{P=p\}\rangle\xrightarrow{\alpha}\langle p',E\cup\{P=p\}\rangle}$	PAR_1^n	$\frac{p\xrightarrow{\alpha}\checkmark}{p\vert\vert q\xrightarrow{\alpha}q}$
PAR_2^n	$\frac{p\xrightarrow{\alpha}p'}{p\vert\vert q\xrightarrow{\alpha}p'\vert\vert q}$	PARC_1^n	$\frac{p\xrightarrow{\alpha}\checkmark\ q\xrightarrow{\beta}\checkmark}{p\vert\vert q\xrightarrow{\alpha\vert\beta}\checkmark}$
PARC_2^n	$\frac{p\xrightarrow{\alpha}p'\ q\xrightarrow{\beta}\checkmark}{p\vert\vert q\xrightarrow{\alpha\vert\beta}p'}$	$\mathrm{PARC}^{\mathrm{n}}{}_4$	$\frac{p\xrightarrow{\alpha}p'\ q\xrightarrow{\beta}q'}{p\vert\vert q\xrightarrow{\alpha\vert\beta}p'\vert\vert q'}$
ALL_1^n	$\frac{p\xrightarrow{\alpha}\checkmark\qquad \alpha\in V\cup\{\tau\}}{allow(V,p)\xrightarrow{\alpha}\checkmark}$	ALL_2^n	$\frac{p\xrightarrow{\alpha}p'\qquad \alpha\in V\cup\{\tau\}}{allow(V,p)\xrightarrow{\alpha}allow(V,p')}$
COM_1^n	$\frac{p\xrightarrow{\alpha}\checkmark}{comm(C,p)\xrightarrow{\gamma_C(\alpha)}\checkmark}$	COM_2^n	$\frac{p\xrightarrow{\alpha}p'}{comm(C,p)\xrightarrow{\gamma_C(\alpha)}comm(C,p')}$
HD_1^n	$\frac{p\xrightarrow{\alpha}\checkmark}{hide(I,p)\xrightarrow{\theta_I(\alpha)}\checkmark}$	HD_2^n	$\frac{p\xrightarrow{\alpha}p'}{hide(I,p)\xrightarrow{\theta_I(\alpha)}hide(I,p')}$

Table 8: SOS $nCRL2$

Definition 4.1.3 (γ_C and θ_I)**.**

$$\gamma_{C_1\cup C_2}(\alpha)=\gamma_{C_1}(\gamma_{C_2}(\alpha))$$

$$\gamma_{\{a_1|\ldots|a_n\to b\}}(\alpha)=\begin{cases} b|\gamma_{\{a_1|\ldots|a_n\to b\}}(\beta) & \textit{if actions } a_i \textit{ occur in } \alpha\ \forall\ 1\leq i\leq n\\ \alpha & \textit{otherwise}\end{cases}$$

$$where\ \beta=\alpha\setminus(a_1|\ldots|a_n).$$

$$\theta_I(\tau)=\tau$$

$$\theta_I(a)=\begin{cases}\tau & \textit{if } a\in I\\ a & \textit{otherwise}\end{cases}$$

$$\theta_I(\alpha_1|\alpha_2)=\theta_I(\alpha_1)|\theta_I(\alpha_2)$$

$nCRL2$ is a language defined to simplify the vast syntax of $mCRL2$ that, even though it allows to express a lot of different behaviours, contains components that are not exploited by the methodology that we are presenting. For this reason, we defined the reduction function that enables the mapping of a $mCRL2$ process term into a $nCRL2$ process term without altering its semantics.

Definition 4.1.4 (Reduction function)**.** *It is a function that given a mCRL2 process term defines its correspondent $nCRL2$ process term removing all that elements that are not considered in $nCRL2$ syntax, like time and data parameters.* $\phi : mCR2_{proc} \cup \{\checkmark\} \rightarrow$ *nCRL2* $_{proc} \cup \{\checkmark\}$

$$\phi(\checkmark) = \checkmark \qquad \phi(\tau) = \tau \qquad \phi(a) = a \qquad \phi(\alpha_1|\alpha_2) = \phi(\alpha_1)|\phi(\alpha_2)$$
$$\phi(a(d_1 \dots d_n)) = undef \qquad \phi(p+q) = \phi(p) + \phi(q) \qquad \phi(p.q) = \phi(p).\phi(q)$$
$$\phi(t >> p) = \phi(p) \qquad \phi(p||q) = \phi(p)||\phi(q) \qquad \phi(P) = P$$
$$\phi(P(t_1 \dots t_n)) = undef \qquad \phi(\Gamma_C(p)) = comm(C, \phi(p))$$
$$\phi(\tau_I(p)) = hide(I, \phi(p)) \qquad \phi(\nabla_V(p)) = allow(V, \phi(p))$$

The correspondence between the two languages is defined by the following propositions.

Proposition 1 (Operational correspondence from $nCRL2$ to $mCRL2$)**.** *Let p be a nCRL2 process*

$$\textit{if } p \xrightarrow{\alpha} p' \textit{ then } \exists\, u \textit{ such that } p \xrightarrow{\alpha}_u q \textit{ and } \phi(q) = p'$$

Proposition 2 (Operational correspondence from $mCRL2$ to $nCRL2$)**.** *Let p be a $nCRL2$ process and $u \in \mathcal{R}^{>0}$*

$$\textit{if } p \xrightarrow{\alpha}_u p' \textit{ then } p \xrightarrow{\alpha} q \textit{ and } \phi(p') = q$$

For the sake of readability the proofs of Propositions 1 and 2 are reported in A.1.

Running example. We illustrate our approach by using, throughout the chapter, a simple travel scenario that is graphically represented, in standard BPMN notation, in Figure 24. The running example includes three participants: the customer, the travel agency and the airline.

In the scenario, a customer sends a flight booking to a travel agent and, upon booking confirmation from the agent, pays and waits for payment confirmation. The travel agent manages in parallel reception of the payment and ordering the flight ticket to an airline company. The airline company evaluates the ticket order and either confirms the payment or refunds the customer.

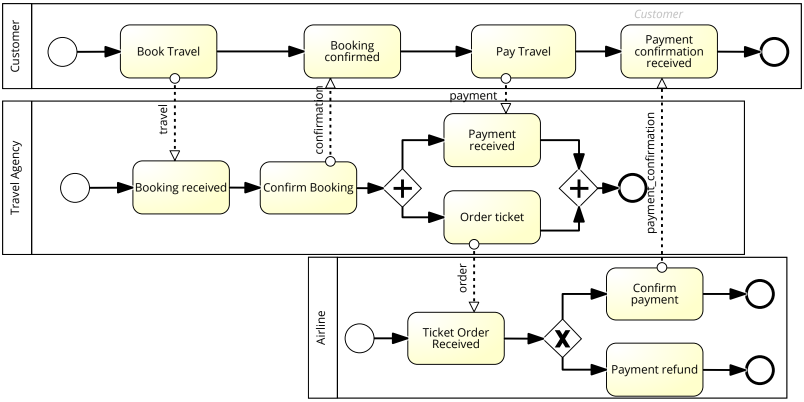


Figure 24: Running example

4.2 PALM Methodology

In this section, we illustrate the PALM methodology outlined in Figure 23. In particular, we describe the *mining* and the *aggregation* steps.

4.2.1 Mining

The mining step is the key part of the PALM methodology since it permits passing from raw data stored in a system log to a formal specification suitable for analysis. This step consists of three phases:

1. *parsing log data;*

2. *mining tool-independent specification;*

3. *transformation into $nCRL2$ specification.*

Preliminaries.

Before going into the details of each phase, we describe the specification language used for describing the intermediate models produced as output in the second phase. Indeed, although we have fully instantiated our proposal for generating $nCRL2$ specifications, we kept the mining process independent from the final target language, by resorting to a *tool-independent description* of the model's structure. This specification language is based on the typical block structure operators of workflow models, and relies on the operators defined by Schimm [115].

Definition 4.2.1 (Block structure syntax)**.**

$$B := a \mid S\{B_i\}_{i \in I} \mid P\{B_i\}_{i \in I} \mid C\{B_i\}_{i \in I} \mid L\{B\}$$

A block structure B is built from *task actions* a by exploiting operators for *sequential composition* (S), imposing an ordered execution of its arguments; *parallel composition* (P), imposing an interleaved execution of its arguments; *exclusive choice* (C), imposing the selection of one block out of its arguments; and *loop* (L), producing an iterative execution of its argument.

Given the above syntax we define the semantics of the block structure language as follows.

Definition 4.2.2 (Block structure semantics)**.** *We define the semantics of a block structure language as a LTS $TS = (S, Act, \rightarrow, s_o)$ where:*

- *S is the set of state in the transition system, that are blocks $b_i \in B$ and one special termination state, denoted by $\checkmark$*

- *Act are the set of labels, i.e. basic actions a plus the silent action τ.*

- *$\rightarrow$ is the transition relation that is inductively defined through operational rules.*

- *s_0 is the initial state.*

op_1	$\frac{}{a \xrightarrow{a} \checkmark}$	
Sop_1	$\frac{B_1 \xrightarrow{a} B_1'}{S\{B_1,\ldots,B_n\} \xrightarrow{a} S\{B_1',\ldots,B_n\}}$	$n \geq 1$
Sop_2	$\frac{B_1 \xrightarrow{a} \checkmark}{S\{B_1,\ldots,B_n\} \xrightarrow{a} S\{B_2,\ldots,B_n\}}$	$n > 1$
Sop_3	$\frac{B \xrightarrow{a} \checkmark}{S\{B\} \xrightarrow{a} \checkmark}$	
Pop_1	$\frac{B_j \xrightarrow{a} B_j'}{P\{B_1,\ldots,B_j,\ldots,B_n\} \xrightarrow{a} P\{B_1,\ldots,B_j',\ldots B_n\}}$	$1 \leq j \leq n$
Pop_2	$\frac{B_j \xrightarrow{a} \checkmark}{P\{B_1,\ldots,B_j,\ldots,B_n\} \xrightarrow{a} P\{B_1,\ldots,B_n\} \backslash \{B_j\}}$	$n > 1$
Pop_3	$\frac{B \xrightarrow{a} \checkmark}{P\{B\} \xrightarrow{a} \checkmark}$	
Cop_1	$\frac{B_j \xrightarrow{a} B_j'}{C\{B_1,\ldots,B_j,\ldots,B_n\} \xrightarrow{a} B_j'}$	
Cop_2	$\frac{B_j \xrightarrow{a} \checkmark}{C\{B_1,\ldots,B_j,\ldots,B_n\} \xrightarrow{a} \checkmark}$	
Lop_1	$\frac{}{L\{B\} \xrightarrow{\tau} \checkmark}$	
Lop_2	$\frac{B \xrightarrow{a} B'}{L\{B\} \xrightarrow{a} S\{B',L\{B\}\}}$	

Table 9: SOS block structure

Parsing log data.

Mining algorithms input an event log and output a model. As already mentioned in Section 2.3, logs are collections of event-based data organised as cases. An event has a name and a lifecycle attribute referring to a state of the transactional lifecycle model of the activity instance producing the event.

In this thesis, we refer to a simplified version of the lifecycle, indicating when an event started and ended using the values *'start'* and *'complete'*, respectively.

We assume that events with the same name and the same attribute of the lifecycle correspond to different executions of the same (unique) system activity.

In the parsing phase of our mining process, each case of the log is transformed into a trace of event names, where the events are ordered according to their completion defined by the 'complete' value of the Lifecycle attribute.

In the (excerpt of the) log in Table 10, concerned with the execution of the Travel Agency component of our running example, since the event 'Confirm booking' starts after the event 'Booking received' has completed, the corresponding trace will include the subtrace 'Booking received, Confirm booking'.

Case	**Event name**	**Lifecycle**
75	Booking received	start
75	Booking received	complete
75	Confirm Booking	start
75	Confirm Booking	complete
75	Payment received	start
75	Order ticket	start
75	Payment received	complete
75	Order ticket	complete
....	....	...

Table 10: Excerpt of Travel Agency log

In this phase, for each trace in the log, we compute a *happened-before relation*, which is used in the next phase. This relation takes into account the chronological order of events and considers only direct dependencies that are given by the lifecycle of the events (and not by the order in the log). This means that an event e is in happened-before relation with an event e' (written $e < e'$) if the completion of e is followed by the starting of e'.

In Table 10, the happened-before relation of case 75 is {Booking re-

Case	Event name	Lifecycle
93	Ticket Order Received	start
93	Ticket Order Received	complete
93	Payment refund	start
93	Payment refund	complete
56	Ticket Order Received	start
56	Ticket Order Received	complete
56	Confirm payment	start
56	Confirm payment	complete
....	....	...

Table 11: Excerpt of Airline log

ceived < Confirm Booking, Confirm Booking < Payment received, Confirm Booking < Order ticket}. Instead, in Table 11, the happened-before relation of case 93 is {Ticket Order Received < Payment refund}, while for case 56 it is {Ticket Order Received < Confirm payment}.

Mining tool-independent specification.

This phase is inspired by the algorithm proposed by Schimm [115]. It consists of seven steps, which manipulate the set of traces in the log to generate the intermediate model described above. The algorithms developed in the thesis retain the name associated with Schimm because the fundamental idea of the algorithm remains closely related to his work. This association helps to establish a conceptual link between the algorithms while acknowledging the modifications made to suit our specific objectives. In particular, the step referred to as "mining tool-independent specification," which closely resembles Schimm's algorithm, utilises the same block structure but incorporates a specific semantics tailored to our requirements instead of relying solely on textual descriptions. Additionally, we introduced a definition of loops necessary for identifying repeated activities and their related patterns within event logs. We also introduced a related metric that quantifies the weight of the loop, which can be leveraged to reduce the state space of the specification. Furthermore, we applied further minimisation techniques to obtain a more compact specification It is important to note that the transformation into an nCRL2 specification and the aggregation steps to obtain the overall specification are not directly related to Schimm's work. Lastly, we provided an implementation of the technique, which is available on GitHub, while Schimm's implementation is not accessible.

Definition 4.2.3 (Loop). *Let E be an event log, $\rho \in E$ a trace, and hb_ρ the happened-before relation of ρ; every loop in ρ starting from event e is identified by a non-empty set of the form $L_e = \{\rho' \subseteq \rho \mid \mathit{first}(\rho') = e\ ,\ (\mathit{last}(\rho') < e) \in hb_\rho\}$, where $\subseteq$ denotes the subtrace relation, and $\mathit{first}(\cdot)$ and $\mathit{last}(\cdot)$ denote the first and last event of a trace, respectively.*

According to the above definition, a loop is identified in a trace ρ when this contains at least a subtrace ρ' such that its last event happened before the first one. Notably, more than one subtrace starting with the same event can have this characteristic, depending on the structure of the body of the loop; hence, all these subtraces are collected together in a set, which will be then analysed to define the structure corresponding to the body of the loop.

The steps of our mining algorithm are the following:

1st step - Search for loops *in: traces, out: traces and sets of subtraces.* All traces retrieved from the log file are analysed in order to identify possible loops. When a subtrace is identified as part of a loop, because its last event is in happened-before relation with the first one (see Def. 4.2.3), the subtrace is replaced by a reference to the loop and stored in the loop set (as in Def. 4.2.3) to be analysed later.
For example, given the log trace *abcdcdf*, after this step we obtain $ab0f$, where 0 is a reference to the loop set $\{cd\}$. From now on, until step 7, we will deal with loop references as events; hence, the happened-before relation of each trace with references will be updated accordingly.

2nd step - Creation of clusters *in: traces, out: traces.* Traces with the same event names and happened-before relations are grouped to form a cluster. This clustering permits reducing the number of traces to process in the following steps, without affecting the structure of the produced model.

For example, given the two traces $abcd$ and $acbd$ with the same happened-before relation $\{a < b,\ a < c,\ c < d,\ b < d\}$, they are unified in the same cluster.

3rd step - Identification and removal of pseudo-dependencies *in: traces and happened-before relation, out: traces.* This step aims at identifying clustered traces that contain pseudo-dependencies, i.e. precedence dependencies between events that are invalidated by other traces. Specifically, given a trace ρ_1 with two events with a dependency of precedence in the happened-before relation of the trace, there should not exist another trace ρ_2 with the same event names in which there is not a relation of precedence between the two events in its happened-before relation. If such other trace ρ_2 exists, then ρ_1 contains a pseudo-dependency and, hence, ρ_1 is removed from the set of trace to be passed to the next step. For example, let us consider a trace corresponding to another case of the log in Table 10 such that its happened-before relation contains the dependency Payment received < Order ticket; this is a pseudo-dependency because the trace corresponding to the case 75 provides the proof that this is not a real dependency; thus it will be discarded.

4th step - Model for each cluster *in: traces, out: set of coarse block structure cluster*. For every cluster of traces we compute the set $\mathcal{P}$ of paths that can be generated by following the happened-before relation. A path is a sequence of events e_1,...,e_n, denoted by $e_1 \rightarrow \ldots \rightarrow e_n$. Notably, a path does not represent a trace, but an ordered sequence of events where each event is in happened-before relation with the next one. Now, every event e will correspond to a basic action in our block structure representation. Every path $p \in \mathcal{P}$, with $p = e_1 \rightarrow \ldots \rightarrow e_n$, is rendered as a sequence block $S\{e_1, \ldots, e_n\}$ (denoted by $S\{p\}$ for short). Thus, a set of paths $\{p_1, \ldots, p_n\}$ is rendered as a parallel block that embeds the sequence blocks corresponding to the included paths, i.e. $P\{S\{p_1\}, ..., S\{p_n\}\}$.

Example 1. *From the cluster:*

{Booking received, Confirm Booking, Payment received, Order ticket}

obtained by the case 75 in Table 10, with happened-before relation:

⟨ Booking received < Confirm Booking, Confirm Booking < Payment received, Confirm Booking< Order ticket⟩

we will obtain the set of paths:

$\mathcal{P}$ = {*Booking received → Confirm Booking → Payment received, Booking received → Confirm Booking → Order ticket* }.

The set $\mathcal{P}$ will result in the following block structure:

$P\{S\{$*Booking received, Confirm Booking, Payment received*$\}, S\{$*Booking received,Confirm Booking,Order ticket*$\}\}$.

5th step - Unify all block structures *in: set of coarse block structures, out: (single) coarse block structure.* All blocks B_1,...,B_n obtained in the previous step are gathered in a single block using the choice operator: $C\{B_1, ..., B_n\}$.

6th step - Restructuring the model *input: coarse block structure, output: block structure.* The structure obtained from the previous step does not yet represent a model of the system behaviour; it is still defined in terms of events rather than actions.

For example, the same event name may appear many times in the model, since it has been generated starting from different cases in the log, but it has to correspond to a single action of the model; such events should be merged into a single one. To this aim, we apply the following transformation rules (the symbol $\rightsquigarrow$ represents a unidirectional transformation from a block structure term to another) up to commutativity of parallel and choice operators:

$$S\{B\} \rightsquigarrow B \qquad C\{B\} \rightsquigarrow B \qquad P\{B\} \rightsquigarrow B$$

$$P\{S\{e, e_1, \ldots, e_n\}, \ldots, S\{e, e'_1, \ldots, e'_m\}\} \rightsquigarrow S\{e, P\{S\{e_1, \ldots, e_n\}, \ldots, S\{e'_1, \ldots, e'_m\}\}\}$$

$$P\{S\{e_1, \ldots, e_n, e\}, \ldots, S\{e'_1, \ldots, e'_m, e\}\} \rightsquigarrow S\{P\{S\{e_1, \ldots, e_n\}, \ldots, S\{e'_1, \ldots, e'_m\}\}, e\}$$

$$C\{S\{e, e_1, \ldots, e_n\}, \ldots, S\{e, e'_1, \ldots, e'_m\}\} \rightsquigarrow S\{e, C\{S\{e_1, \ldots, e_n\}, \ldots, S\{e'_1, \ldots, e'_m\}\}\}$$

$$C\{S\{e_1, \ldots, e_n, e\}, \ldots, S\{e'_1, \ldots, e'_m, e\}\} \rightsquigarrow S\{C\{S\{e_1, \ldots, e_n\}, \ldots, S\{e'_1, \ldots, e'_m\}\}, e\}$$

The rules are syntax driven; in the case in which more than one rule can be applied, an arbitrary order is defined.

Example 2. *By applying these rules to the block*

$C\{P\{S\{$*Booking received, Confirm Booking, Payment received*$\}, S\{$*Booking received,Confirm Booking,Order ticket*$\}\}\}$

we obtain the block:

$S\{$*Booking received, Confirm Booking, P*$\{$*Payment received,Order ticket*$\}\}$

7th step - Replacing loop references *in: loop sets, out: block structure.* In this step, we rerun the algorithm over the traces in the loop sets until no loop block exists in the traces. In this way, we obtain a block structure B for each loop set; the term $L\{B\}$ will then replace all occurrences of the corresponding reference.
For example, the trace shown in the first step results in the block $S\{a, b, 0, f\}$, that after this step becomes $S\{a, b, L\{S\{c, d\}\}, f\}$.

The technical details concerning each step of the mining algorithm can be found in Appendix B, which provides comments on the source code of the implementation [97], complete results of the running example in Figure 24 and how to run the experiments executed in Section 4.4.

Transformation into $nCRL2$ specification.

The previous phase gives as output a block structure specification that is independent from a specific analysis tool. This choice makes the mining process flexible to be extended to produce specifications written in different languages, to exploit different process-algebraic techniques and tools.

Here, to demonstrate feasibility and effectiveness of our proposal, we have targeted the methodology to $nCRL2$ specifications, i.e. a formal specification language that guarantees operational correspondence to $mCRL2$ (as shown in Section 4.1). To obtain a $nCRL2$ specification, we defined a function $\mathcal{T} : \mathbb{B} \rightarrow \langle p, E\rangle$ where $\mathbb{B}$ is the set of block structures and $\langle p, E\rangle$ is a $nCRL2$ process specification. Intuitively, the transformation function inputs a block structure and outputs a pair composed of a $nCRL2$ process and a related set of process definitions.

Formally, function $\mathcal{T}$ is defined inductively on the syntax of block structures as follows:

$\mathcal{T}(\checkmark) = \checkmark$
$\mathcal{T}(\tau) = \langle \tau, \emptyset \rangle$
$\mathcal{T}(a) = \langle a, \emptyset \rangle$
$\mathcal{T}(S\{B_i\}_{i\in I}) = \langle \cdot_{i\in I}\mathcal{T}(B_i), E \rangle$
$\mathcal{T}(C\{B_i\}_{i\in I}) = \langle +_{i\in I}\mathcal{T}(B_i), E \rangle$
$\mathcal{T}(P\{B_i\}_{i\in I}) = \langle ||_{i\in I}\mathcal{T}(B_i), E \rangle$
$\mathcal{T}(L\{B\}) = \langle K, \{K = (\mathcal{T}(B).K + \mathcal{T}(B))\} \cup E \rangle$ with K fresh

A valid termination ($\checkmark$) is a special process that has been added to the syntax of processes in order to identify a good termination; this definition is valid both for the block structure and the $nCRL2$ language.

τ and visible actions are straightforwardly transformed respectively into τ and $nCRL2$ actions, without producing any process definition. Each composition structure operator, except for the loop one, is rendered in terms of the corresponding $nCRL2$ operator: S as ., C as $+$, and P as $||$. Thus, a sequential composition of blocks is transformed into a pair, where the first element is a sequential composition of the processes resulting from the transformation of each inner block, and the second element is the set given by the union of the process definitions resulting from the transformation of each inner block. The transformation of choice and parallel composition are similar.

Instead, a loop structure is rendered as a pair whose first element is a process call with a fresh identifier K and whose second element is the union of the recursive definition of K with the process definitions resulting from the transformation of the inner block. The definition of K is given in terms of the process resulting from the transformation of the block occurring as body of the loop; it ensures the execution of at least one iteration of the body.

The correspondence between a block structure and the $nCRL2$ process is stated as follows, where we mark the operational semantics of $nCRL2$ as $\rightsquigarrow$ to diversify it from the operational semantic of the block structure

Proposition 3 (Operational correspondence from Block Structure to $nCRL2$)**.** *Given a block structure B it holds that*

$$\textit{if } B \xrightarrow{a} B' \textit{ then } \mathcal{T}(B) \overset{a}{\rightsquigarrow} \mathcal{T}(B'') \textit{ and } \mathcal{T}(B') = \mathcal{T}(B'')$$

Proposition 4 (Operational correspondence from $nCRL2$ to Block Structure)**.** *Given a block structure B it holds that*

$$\textit{if } \mathcal{T}(B) \overset{a}{\rightsquigarrow} \mathcal{T}(B') \textit{ then } B \xrightarrow{a} B'' \textit{ and } \mathcal{T}(B') = \mathcal{T}(B'')$$

The complete demonstration of Propositions 3 and 4 is in Appendix A.2. A pair $\langle P, \{K_1 = P_1, \ldots, K_n = P_n\}\rangle$ produced by $\mathcal{T}$ corresponds to the following $nCRL2$ specification (we use notation $act(\cdot)$ to indicate the actions occurring within a term of a specification):

```
act
act(P), act(P1), ... , act(Pn);
proc
K=P; K1=P1; ... ; Kn=Pn;
init K;
```

Example 3. *If we apply the $\mathcal{T}$ function to the block structure resulting from the log in Table 10 (since the example does not contain loops, for the sake of readability we omit the second element of the pair generated from $\mathcal{T}$), we obtain:*

$$\begin{aligned}&\mathcal{T}(S\{\textit{Booking received,Confirm Booking,}P\{\textit{Payment received,Order Ticket}\}\})\\ &= \textit{Booking received.Confirm Booking.(Payment received } || \textit{ Order Ticket)}\end{aligned}$$

From the block structure resulting from the log in Table 11 we obtain:

$$\begin{aligned}&\mathcal{T}(S\{\textit{Ticket Order Received,}C\{\textit{Payment Refund,Confirm payment}\}\})\\ &=\textit{Ticket Order Received.(Payment refund + Confirm payment)}\end{aligned}$$

Using the transformation function $\mathcal{T}$ we have obtained an $nCRL2$ process specification well defined from the process algebraic point of view, i.e. it respects the syntax of the $nCRL2$ language given in Section 4.1. However, in this actual form, the specification cannot be used as input for the analysis tools provided by the $mCRL2$ toolset. Indeed, these tools require the $mCRL2$ specification and, consequently, the $nCRL2$ specification, also to respect the pCRL format [107], where parallel, communication, renaming and hiding operators must be positioned at top level.

Therefore, we have defined another function, $\mathcal{T}_p$, to transform a process specification produced by $\mathcal{T}$ (possibly with parallel operator at any level of nesting, and not using communication, renaming and hiding operators) into an equivalent one in the pCRL format. Formally, $\mathcal{T}_p$ takes as input a pair $\langle P, D\rangle$ and returns a tuple $\langle P', D', CommSet, AllowSet, HideSet\rangle$, where P' and D' are a process and a set of process definitions where the parallel operator is moved at top level, while $CommSet$, $AllowSet$ and $HideSet$ are sets of communication expressions, allowed actions and hidden actions, respectively. Intuitively, to move nested parallel processes to the top level, the $\mathcal{T}_p$ function uses additional synchronisation actions that permit to properly activate the moved processes and to signal their termination. These added actions are forced to communicate and the actions resulting from their synchronisations are hidden.

For the sake of presentation, to avoid dealing with projections and other technicalities concerning tuples, we provide in Figure 25 a simplified definition of $\mathcal{T}_p$ in which we do not explicitly represent the sets of communication expressions, allowed actions and hidden actions; such sets are indeed populated (in a programming style) by means of functions $addComm$, $addAllow$ and $addHide$, respectively. The sets $CommSet$ and $HideSet$ are instantiated to $\emptyset$, while the set $AllowSet$ is instantiated to the set of all actions of the process and the process definitions to be transformed. We use t, t_i and t_h to denote the synchronisation actions. In case of process definitions, it is not sufficient to move the parallel operator at top level of the process occurring as body; the operator has to be removed by expanding the term according to the interleaving semantics of the operator (like CCS's expansion law [88, Sec. 3.3]). Specifically, a process definition $K = P$ is transformed into $K = \mathcal{T}_d(P)$, where the auxiliary function $\mathcal{T}_d$ is defined as follows:

$$\mathcal{T}_d(a) = a \qquad \mathcal{T}_d(\cdot_{i\in I} P_i) = \cdot_{i\in I}\mathcal{T}_d(P_i) \qquad \mathcal{T}_d(+_{i\in I} P_i) = +_{i\in I}\mathcal{T}_d(P_i)$$
$$\mathcal{T}_d(||_{i\in I} P_i) = +_{s\in(\bigcup_{i\in I} seq(\mathcal{T}_d(P_i)))} s \qquad \mathcal{T}_d(K) = K$$

with function $seq(P)$ returning the set of all sequences of actions/calls of P.

With $\mathcal{T}_{seq}$, each process in the parallel block is surrounded by a pair of synchronisation actions (t and t) that can communicate only after that the sequence preceding the parallel process has been executed. For example, $a.(b||c)$ turns into $allow(\{t', a, b, c\}, comm(\{t|t \rightarrow t'\}, a.t.b.t||t.c.t))$. With $\mathcal{T}_{ch}$ we surround each process in the choice with a pair of synchronisa-

$$
\begin{aligned}
&\mathcal{T}_p(a) = a\\
&\mathcal{T}_p(\cdot_{i\in I} P_i) = \mathcal{T}_{seq}(\cdot_{i\in I} \mathcal{T}_p(P_i))\\
&\mathcal{T}_p(+_{i\in I} P_i) = \mathcal{T}_{ch}(+_{i\in I} \mathcal{T}_p(P_i))\\
&\mathcal{T}_p(||_{i\in I} P_i) = ||_{i\in I} \mathcal{T}_p(P_i)\\
&\mathcal{T}_p(K) = K
\end{aligned}
$$

$$
\mathcal{T}_{seq}(\cdot_{i\in\{1,\dots,n\}} P_i) =
\begin{cases}
\mathcal{T}_{seq}\big((\cdot_{i\in\{1,\dots,j-1\}} P_i).t.Q_1.t.(\cdot_{h\in\{j+1,\dots,n\}} P_h)\big)|| & \text{if } \exists j \in I : \\
||_{m\in M\setminus\{1\}} t.Q_m.t & \quad P_j = ||_{m\in M} Q_m \\
\quad \text{with } addComm(t(|t)^{|M|-1} \to t'), & \wedge\ t \text{ and } t' \text{ fresh} \\
\qquad addAllow(t'), addHide(t') & \\
 & \\
\cdot_{i\in\{1,\dots,n\}} P_i & \text{otherwise}
\end{cases}
$$

$$
\mathcal{T}_{ch}(+_{i\in I} P_i) =
\begin{cases}
\mathcal{T}_{ch}\big((+_{i\in I\setminus\{j\}} t_i.P_i.t_i) + t.Q_1.t\big)|| & \text{if } \exists j \in I : \\
||_{m\in M\setminus\{1\}}((+_{h\in I\setminus\{j\}} t_h.t_h) + t.Q_m.t) & \quad P_j = ||_{m\in M} Q_m \\
\text{with } addComm(\{t(|t)^{|M|-1} \to t'\} & \wedge\ t \text{ fresh} \\
\qquad\qquad \cup \{t_i(|t_i)^{|M|-1} \to t' \mid i \in I \setminus \{j\}\}), & \wedge\, \forall_{i\in I\setminus\{j\}} t_i \text{ fresh} \\
\qquad addAllow(t'), addHide(t') & \\
 & \\
+_{i\in I} P_i & \text{otherwise}
\end{cases}
$$

Figure 25: Definition of function $\mathcal{T}_p$ (and related auxiliary functions).

tion actions that are then used in the new parallel process to give the same possibility of executing a choice among processes in each process inside it. For example, $a + (b||c)$ turns into $allow(\{t', a, b, c\}, comm(\{t|t \to t', t1|t1 \to t'\}, t.a.t + t1.b.t1||t.t + t1.c.t1))$.

Example 4. *If we apply function $\mathcal{T}_p$ to $\langle P, \emptyset\rangle$, where P is the first $nCRL2$ process produced in Example 3 (the second one does not contain the parallel operator). We obtain $\mathcal{T}_p(\langle P, \emptyset\rangle) = \langle \mathcal{T}_p(P), \emptyset, CommSet, AllowSet, HideSet\rangle$, where:*

$$\begin{aligned}\mathcal{T}_p(P) &= \mathcal{T}_{seq}(\mathcal{T}_p(\textit{Booking received}).\mathcal{T}_p(\textit{Confirm Booking}).\\ &\qquad (\mathcal{T}_p(\textit{Payment received}) \mid\mid \mathcal{T}_p(\textit{Order Ticket})))\\ &= \textit{Booking received.Confirm Booking.t.Payment received.t} \mid\mid \textit{t.Order Ticket.t}\end{aligned}$$

$$\begin{aligned}AllowSet = \{&t', \textit{Booking received, Confirm Booking, Payment received,}\\ &\textit{Order Ticket}\}\end{aligned}$$

$$CommSet = \{t|t \rightarrow t'\} \qquad HideSet = \{t'\}$$

Thus, a tuple $\langle P, \{K_1 = P_1, \ldots, K_n = P_n, CommSet, AllowSet, HideSet\rangle$ produced by $\mathcal{T}_p$ corresponds to the following $nCRL2$ specification:

```
act
act(AllowSet), act(CommSet);
proc
K=P; K_1=P_1; ... ; K_n=P_n;
init hide(HideSet,allow(AllowSet,comm(CommSet,K)));
```

4.2.2 Aggregation

In this step, the specifications obtained from the logs of components of a distributed system can be combined to obtain an aggregate specification of the overall system. This allows one to focus analysis on the overall behaviour of a system resulting from the message-based interactions among its components. This step takes advantage of the parallel composition operators that enable channel-based communication, to obtain the specification of the full system.

To enable the aggregation step, it is necessary to extract from the logs the information concerning message exchanges. This information is specified in the events stored in XES logs by specific attributes indicating input and output messages. Below, we report the XES code corresponding to an event associated to the 'Booking received' task, which receives a 'travel' message:

```
<event>
  <string key="concept:name" value="Booking received"/>
  <string key="input_message" value="travel"/>
  <string key="lifecycle:transition" value="start"/>
  <date key="time:timestamp" value="2020-07-01T01:03:10+01:00"/>
</event>
```

Information about message exchanges is extracted from the logs during the parsing phase, and is made available to the aggregation step in terms of two partial functions: M_{inp} (resp. M_{out}) takes as input an event name and returns the name of the received (resp. sent) message, if any.

We define how an aggregate specification is obtained below; we use notation $cod(\cdot)$ to indicate the codomain of a function.

Definition 4.2.4 (Aggregation)**.** *Let $\langle P_i, D_i, CommSet_i, AllowSet_i, HideSet_i\rangle$, with $i \in I = \{1, \ldots n\}$, be specification tuples obtained at the mining step, and M_{inp} and M_{out} be input and output message functions; the sets defining their aggregate specification are as follows:*

- $Act_i \ = \ act(P_i) \cup act(D_i)$, *with* $i \in I$;
- $Act_{agg} \ = \ \bigcup_{i \in I}(Act_i \cup act(CommSet_i)) \cup cod(M_{inp}) \cup cod(M_{out})$;
- $CommSet_{agg} \ = \ \bigcup_{i \in I} CommSet_i \ \cup$ $\{a_1|a_2 \ \rightarrow \ m \ \mid \ a_1 \ \in \ Act_i, \ a_2 \ \in \ Act_j, \ i \ \neq \ j, \ M_{inp}(a_1) \ = \ M_{out}(a_2) = m\}$;
- $AllowSet_{agg} \ = \ \bigcup_{i \in I} AllowSet_i \ \cup \ cod(M_{inp}) \ \cup \ cod(M_{out}) \ \setminus$ $\{a_1, a_2 \mid a_1 \in Act_i, \ a_2 \in Act_j, \ i \neq j, \ M_{inp}(a_1) = M_{out}(a_2)\}$;
- $HideSet_{agg} \ = \ \bigcup_{i \in I} HideSet_i$.

Hence, the corresponding aggregate specification is:

```
act
Act_agg
proc
K_1=P_1;  ...;  K_n=P_n;  D_1;  ...;  D_n
init hide(HideSet_agg,allow(AllowSet_agg,comm(CommSet_agg,K_1||...||K_n)));
```

Every time two events correspond to a message exchange between two tasks, the communication is described as a synchronisation of actions, which results in an action named with the message name.

We conclude with a simple example aiming at clarifying the aggregation step; a richer example based on the running scenario is provided in the next section.

Example 5. *Let us consider a simple collaborating scenario where one participant sends a message m_1 to another one and then waits for a series of messages m_2; on the other side, after receiving the message m_1, the participant decides either to perform an internal activity and stop, or to perform a different internal* *activity and send a series of messages m_2. This behaviour is captured by the mining step in terms of the following specification tuples:*

$$\langle a.K_1, \{K_1 = (b.K_1 + b)\}, \emptyset, \{a, b\}, \emptyset\rangle$$
$$\langle c.(d + (e.K_2)), \{K_2 = (f.K_2 + f)\}, \emptyset, \{c, d, e, f\}, \emptyset\rangle$$

and the functions providing the messages information extracted from the logs are defined by the following cases:

$$M_{out}(a) = m_1,\ M_{out}(f) = m_2,\ M_{inp}(c) = m_1, \text{ and } M_{inp}(b) = m_2.$$

Now, the sets defining the corresponding aggregate specification are defined as follows:

$$\begin{aligned}
&Act_{agg} = \{a, b\} \cup \{c, d, e, f\} \cup \{m_1, m_2\}\\
&CommSet_{agg} = \{a|c \rightarrow m_1, f|b \rightarrow m_2\}\\
&AllowSet_{agg} = Act_{agg} \setminus \{a, c, f, b\} = \{d, e, m_1, m_2\}\\
&HideSet_{agg} = \emptyset
\end{aligned}$$

The resulting aggregate specification is as follows:

```
act
a, b, c, d, e, f, m1, m2
proc
K3=a.K1;  K4=c.(d + (e.K2));  K1 = (b.K1 + b);  K2 = (f.K2 + f)
init  allow({d, e, m1, m2},comm({a|c → m1, f|b → m2},K3||K4));
```

where the hide command is omitted since the hiding set is empty.

4.3 PALM at Work

The PALM methodology introduced in the previous section has been implemented as a command-line Java tool, called PALM as well, whose source and binary code is available on GitHub [97].

The PALM tool enables us to analyse both the specification resulting from a single event log and the aggregate specification resulting from multiple logs. To support the analysis of the produced specifications, the tool provides an interface for some of the most used $mCRL2$ tools (mcrl22lps, lps2lts, etc.). It also provides additional functionalities to support the validation illustrated in Section 4.4, such as the computation of the fitness measure to analyse the quality of the obtained specifications, and the transformation into the $nCRL2$ language of the models produced by other process mining algorithms, to compare their outcome with the specifications produced by PALM.

In addition, to keep the state space of the produced specifications manageable for the analysis, the tool allows users to set a *loop threshold* parameter, which is used during the generation of the $nCRL2$ specification to decide whether to unfold a loop or to represent it as a process definition. Such a decision is taken by comparing the value of this parameter with the frequency value computed for each loop in the block structure specification. The loop frequency measures the weight of a loop considering how many times this loop appears in the log and its length. This value ranges between 0 to 100, where 100 means that the loop has high relevance in the log, i.e. every trace in the log is produced by the loop, while 0 means that the loop's events do not appear in the log. Thus, if t is the frequency threshold chosen by the user, the PALM tool will write as recursive processes only those loops that have frequency greater than or equal to t, while all the other loops are unfolded according to their frequency. We defined loop frequency in PALM as follows.

Definition 4.3.1 (Loop frequency). *Given a loop l and a set $\{c_i\}_{i \in I}$ of cases of a log, the frequency of the loop is computed as follows:*

$$f_{loop}(l, \{c_i\}_{i \in I}) = \left(\frac{\sum_{i \in I} f_{loop}(l, c_i)}{n_{cases}} + \frac{n_{cases} \times 100}{|I|} \right) /2$$

where l is the loop, n_{cases} is the number of cases in $\{c_i\}_{i \in I}$ in which l is present. The frequency over a single case is computed as follows:

$$f_{loop}(l, c) = occ(l, c) \times |l| \ / \ |c| \times 100$$

where $occ(l, c)$ returns the number of occurrences of the loop l in c, while $|c|$ (resp. $|l|$) returns the length of c (resp. l).

We conclude the section with the application of the PALM methodology and its tool to our running example.

Example 6. *Consider the running example in Figure 2. To apply the bottom-up approach to this running example, synthetically generated event logs corresponding to each participant (i.e., Space Agency, Data Centre) were necessary. Since the example involves a collaboration scenario, the aggregation capability of the PALM tools was then used to generate the overall specification as follows.*

```
act
S1,S2,S3,S4,S5,S6,S7,D1,D2,D3,t0,t,share,result;
proc
S=(S1.S2.t0.S3.t0.S6.S7)||(t0.S4+S5.t0);
D=D1.D2.D3;
init
hide({t},
    allow({t,share,result,S1,S3,S4,S5,S7,D2},
        comm({t0|t0->t,S2|D1->share,D3|S6->result},
            S||D)));
```

Listing 4.1: $nCRL2$ aggregated specification of the running example in Figure 2.

The specification allows checking properties related to the overall specification, such as deadlock freedom[1], which is satisfied, or other specific properties of the model, e.g. it should always be possible to access the Data Centre help for computation of satellite collision $[true * .D2.true*]true$, that results in a satisfied property for the specification.

Example 7. *Let us now consider an example where the collaboration condition is not satisfied. The example starts with the $nCRL2$ process specifications obtained (separately) from the event logs corresponding to each participant of our running example (i.e., Customer, Travel agency and Airline). They correspond to processes P0, P1 and P2 in Listing 4.2 (their full $nCRL2$ specifications are reported in [19]).*

When these specifications are analysed with mCRL2 tools, the individual process behaves as expected (e.g., no deadlock occurs - all states of the LTS corresponding to the specification have outgoing transitions). However, since they are specifications of components of a single distributed system, it is important to check also their aggregate specification in Listing 4.2.

```
act
Confirmpayment,BookTravel,Bookingreceived,Paymentreceived,Paymentrefund,
Bookingconfirmed,ConfirmBooking,confirmation,Orderticket,
TicketOrderReceived,PayTravel,t,Paymentconfirmationreceived,payment,
payment_confirmation,t0,order,travel;
proc
P0=(TicketOrderReceived.(Paymentrefund+Confirmpayment));
P1=(BookTravel.Bookingconfirmed.PayTravel.Paymentconfirmationreceived);
P2=((Bookingreceived.ConfirmBooking.t0.Paymentreceived.t0)
    ||(t0.Orderticket.t0));
```

[1] In the deadlock checking, the mCRL2 tool is not able to distinguish between a correct termination and an actual deadlock. Anyway, since the $Terminate$ action is appended to each correct termination, we can solve this issue by resorting to the model checking of the logical formula $[!Terminate*] < true > true$.

```
init
hide({t},allow({Paymentrefund ,confirmation ,t ,payment ,payment_confirmation ,
                    order ,travel},
  comm({Bookingreceived|BookTravel->travel ,
          Confirmpayment|Paymentconfirmationreceived->payment_confirmation ,
          Orderticket|TicketOrderReceived->order ,
          PayTravel|Paymentreceived->payment ,
          Bookingconfirmed|ConfirmBooking->confirmation ,t0|t0->t},
     P0||P1||P2)));
```

Listing 4.2: $nCRL2$ aggregate specification of the running example.

The fact that the Customer will wait forever to receive the payment confirmation if the Airlane has to refund the payment (see Figure 24) is observable only having the overall specification, since when analysed separately there is no communication between the participants. Using one of the mCRL2 checking functionality, we detect a deadlock. Interestingly, the tool, in case of deadlock, offers a counterexample trace, i.e. the ordered sequence of actions that leads to the deadlocked state. In our example, it reports a trace where the customer has paid for the travel, the order is sent by the travel agency to the airline company, but the latter takes the "Payment refund" choice and the customer process waits forever the "payment_confirmation" message.

4.4 Validation

In this section, we report the results of the experiments we carried out to empirically validate the PALM methodology and the related tool, considering both logs synthetically generated using PLG2 [28] and logs from real scenarios [1].

Validation Overview.

In the validation we compare the results of the experiments conducted with PALM against those obtained by using three well-known process mining discovery algorithms presented in Section 2.3, i.e. Inductive Miner (IM), Structured Heuristic Miner (S-HM), and Split Miner (SM), supported by the TKDE Benchmark tool discussed in [14]. We consider these algorithms since they perform quite well in terms of mining time and, also, perform better than others in terms of quality measures [14].

Our comparison is based on a revised version of the *fitness* quality measure used in process mining to evaluate discovery algorithms [26, 109]. Like the original fitness measure, also our notion aims at measuring the proportion of behaviour in the event log that is in accordance with the model but does this differently by taking advantage from the model checking technique enabled by our process algebraic specifications. For this reason, we refer to it as *model checking-based fitness*.

In this work, we focus on fitness since it is the measure most considered in the literature; we leave as future investigation the introduction of other quality measures from the process mining field, namely precision, to quantify how much a process model overapproximates the behaviour seen in an event log [122], and generalisation, to assesses the extent to which the resulting model will be able to reproduce future behavior of the process [26].

Definition 4.4.1 (Model Checking-based fitness). *Let C be the set of cases of a log and S be an $nCRL2$ specification, the Model Checking-based fitness (MC-fitness) measures the ability of the specification S to satisfy the formulas f_c such that $c = [e_1, ..., e_n] \in C$ and $f_c =< tau^*.e_1.tau^*. \cdots .tau^*.e_n.tau^* > true$. The MC-fitness is computed as follows:*

$$\textit{MC-fitness}(C, S) = \ |\{f_c \mid c \in C \ , \ S \models f_c\}| \ / \ |C|$$

where $S \models f_c$ indicates that the formula f_c is satisfied by the specification S.

Notably, f_c is a formula describing the case of a log, where each case event is surrounded by an unbounded number of silent actions. Formulas are verified using $mCRL2$ model checker. The values of MC-fitness range from 0 to 1, with 1 meaning that every formula can be satisfied, and 0 that none of them can.

Validation has been also enriched by checking equivalence of process models resulting from the PALM technique and those obtained from the three process mining algorithms. The considered equivalences are those supported by the $mCRL2$ tool: strong bisimilarity, weak bisimilarity, trace equivalence, weak trace equivalence, branching bisimilarity, strong simulation and divergence preserving branching bisimilarity [65]. This part of the validation is interesting because it permits to detect those situations where two techniques have similar fitness values but yield different models from the behavioural point of view (i.e., they are not equivalent up to any equivalence relation).

Validation Set-up.

Figure 26 describes the preparatory steps needed for comparing two different kinds of models, i.e. a process algebra specification with a BPMN model.

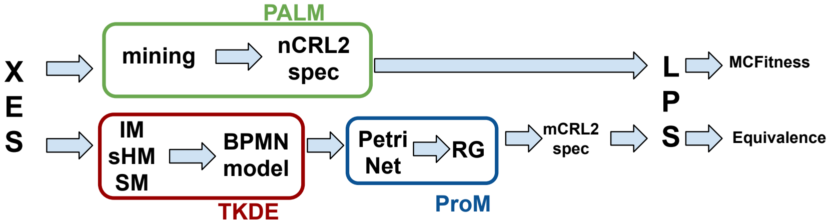


Figure 26: Transformation steps required by the PALM validation.

To make such comparison, we resort to a common specification model, that is LPS (Linear Process Specifications) and transform the BPMN models, obtained by executing the three considered process mining algorithms via the TKDE tool [14] according to the following steps.

We then use ProM [45], a well-established framework that supports a wide variety of process mining techniques; in particular we use two of its plug-ins, namely "Convert BPMN diagram to Petri net" and

"Construct reachability graph of a Petri net". The BPMN models are first transformed into Petri Nets, and then their Reachability Graph (RGs) are obtained. From the RG it is straightforward to obtain $mCRL2$ specifications (Definition 4.4.2) below. $mCRL2$ specifications are given as input to the appropriate $mCRL2$ tool to be transformed into LPS, which can be used to calculate the MC-fitness and run conformance checking.

Definition 4.4.2 (From Reachability Graph to mCRL2). *Let $\langle E, M \rangle$ be a reachability graph, where E is the set of edges of the form $< v, l, v' >$ with $v, v' \in V$, $l \in L$, while $M \subseteq V$ is the initial marking representing the initial distribution of tokens. V is the set of vertices and L is the set of labels. The corresponding mCRL2 specification is as follows:*

Model Name	Discovery Algorithm		Mining Time (s)	MC Fitness	Equiv.
log1	PALM	90	$\leqslant 1$	1	-
		50	$\leqslant 1$	1	-
		0	$\leqslant 1$	1	-
	IM		59,2	1	weak-trace
	sHM		88,2	1	weak-trace
	SM		12,4	1	weak-trace
log2	PALM	90	$\leqslant 1$	1	-
		50	$\leqslant 1$	1	-
		0	$\leqslant 1$	1	-
	IM		101,2	1	branching-bisim
	sHM		163,4	1	none
	SM		24,4	0.78	none
log3	PALM	90	$\leqslant 1$	0,87	-
		50	$\leqslant 1$	0,87	-
		0	$\leqslant 1$	0,87	-
	IM		127,4	0,99	none
	sHM		129	0,99	none
	SM		23,8	0.99	none
rlog1	PALM	90	9,5	0,5	-
		50	4,96	0,5	-
		0	$\leqslant 1$	0,6	-
	IM		21	1	none
	sHM		73,2	0	none
	SM		42,6	N.C.	N.C.
rlog2	PALM	90	$\leqslant 1$	0,4	-
		50	$\leqslant 1$	0,4	-
		0	$\leqslant 1$	0,4	-
	IM		16,4	N.C.	N.C.
	sHM		154	0	none
	SM		59,6	N.C.	N.C.
rlog3	PALM	90	$\leqslant 1$	0,66	-
		50	$\leqslant 1$	0,66	-
		0	3	N.C.	-
	IM		23,2	1	none
	sHM		41	0	none
	SM		22,7	N.C.	N.C.
rlog4	PALM	90	$\leqslant 1$	0,85	-
		50	$\leqslant 1$	0,85	-
		0	$\leqslant 1$	0,85	-
	IM		6,4	1	none
	sHM		43,8	0,8	none
	SM		53,6	0,7	none
rlog5	PALM	90	$\leqslant 1$	0,71	-
		50	$\leqslant 1$	0,85	-
		0	$\leqslant 1$	0,85	-
	IM		6,4	N.C.	N.C.
	sHM		56,4	0,71	none
	SM		23,0	0,42	none
rlog6	PALM	90	$\leqslant 1$	0,77	-
		50	$\leqslant 1$	0,83	-
		0	$\leqslant 1$	N.C	-
	IM		27,6	1	none
	sHM		70	0	none
	SM		18,8	0,22	none

Table 12: Results of the PALM Validation.

```
act
$L$
proc
$\{K_v = l_1.K_{v_1} + \cdots + l_k.K_{v_k} \mid < v, l_1, v_1 >, \ldots, < v, l_k, v_k > \in E\}$
$\cup \{K_v =$ delta $\mid \nexists < v, l, v' > \in E\}$
init $||\{K_v \mid v \in M\}$;
```

where delta *is the special mCRL2 process that cannot perform anything, and* $||\{K_i\}_{i\in\{1,\ldots,n\}}$ *denotes the term* $K_1|| \ldots ||K_n$.

Validation Results.

Table 12 summarises the validation that we ran over three synthetically generated event logs (whose generating models are publicly available in [97]) and six real-life event logs.

All the synthetic logs (log1, log2, log3) are built out of 1000 cases that mix parallel and choice behaviours. Specifically, log1 is generated by a BPMN model with two XOR gateways (split and join) and with six tasks, while log2 is generated by a model with four gateways (XOR and AND with split and join) and nine tasks. Differently, log3 is generated by a model with eight gateways (six XOR, two AND split and join), fourteen tasks and also includes two loops.

The real-life logs (rlog1, rlog2, rlog3, rlog4, rlog5, rlog6) refer to activities of daily living performed by several individuals and collected by using sensors. The logs can be retrieved online [97] as an extraction of what data.4tu makes available[2].

All logs are given as input to PALM and to the other discovery algorithms. For each of them, we register the mining time to generate the specification (in seconds), and the value of the MC-fitness. We also calculate if there exists an equivalence relation between the model generated by PALM and the ones generated by IM, sHM and SM. In the discovery algorithm column, for the rows related to PALM, we also specify the loop frequency values (defined in Def. 4.3.1), i.e. 90, 50 and 0. The symbol $-$ used in the Table 12 means that the value of that cell does not need to be computed, while the value $N.C.$ means that we tried to calculate it but a timeout expired.

According to our experimentation, PALM behaves quite well with the synthetic logs. In particular, the application of the PALM methodology to log1 returns an MC-fitness equal to 1. When comparing the three models generated by PALM (with different threshold) and those obtained from the same log by the other algorithms, we can observe that weak-trace equivalence is satisfied. This means that all resulting models can produce the same cases, possibly with a different number of silent actions.

[2] `https://data.4tu.nl/repository/uuid:01eaba9f-d3ed-4e04-9945-b8b302764176`

Considering log2, the comparison does not change much, apart from the observed equivalence. In this case, even if we obtain the same value for the MC-fitness, the models are not equivalent up to any of the considered relations. This is because the generated models can reproduce cases not included in the current log. For log3, instead, we do not have a perfect MC-fitness value; this is probably due to the difficulty to properly identify those situations where there is a choice between performing a task and skipping it, this is due to the minimisation applied during the model generation.

The experiments with real logs confirm that there is no equivalence relation according to which the four models are equivalent, but we have quite different results for fitness. Since fitness values are so different from each other, it is straightforward that no equivalence exists between the generated models. Hence, let us focus more on the fitness results.

PALM generates from logs rlog1 and rlog2 two specifications with a value of fitness not high, which anyway is in line with the other discovery algorithms (actually, the sHM algorithm generates a model that is not able to reproduce any case in the log). Logs rlog3 and rlog6 show the importance of the loop frequency threshold parameter for real logs, where the number of loops causes a state-space explosion. Unfolding the 'less important' loops, i.e. the loops with a low loop frequency, allows us to complete the analysis over specifications which could not be treated in the standard way. For logs rlog4 and rlog5, our mining tool performs better than the others, as the specifications generated by PALM are able to reproduce most of the cases in the logs, while in rlog1, rlog3 and rlog6 IM outperforms PALM and the other algorithms in terms of MC fitness. In terms of time for generating the models, PALM always outperforms all the other algorithms.

4.5 Related Work

In the literature, there is other work that pursuits the goal of generating models from a set of observations. Such research topic is investigated by both the process mining community and the engineering and formal methods community.

On Process Mining. PALM differentiate itself from different process discovery techniques available in the literature, e.g. [131, 12, 136, 13, 77] for the following aspects. It focuses on collaborative systems, while the mentioned techniques primarily concentrate on single organizations and organizational improvement. PALM specifically targets collaborative systems, it is designed to handle scenarios where multiple organizations collaborate, communicate, and share information to achieve common goals. It handles distribution and communication aspects. Unlike other techniques that use specification languages like Petri Nets or Process Trees, PALM is specifically developed to address the challenges of distribution and communication aspects in collaborative systems while other languages often struggle to represent distributed behaviour effectively. It is effective in composing distributed behavior, PALM's methodology aims at capturing and representing distributed behaviour in collaborative systems. It can discover process algebraic specifications tailored to such scenarios, making it suitable for handling complex collaboration scenarios. Finally, PALM provides a way to verify collaboration properties using formal methods. As an exception, in [35] BPMN collaborations are discovered; however, differently from [35] the PALM approach defines a specific process mining algorithm capable of discovering process algebraic specifications of collaborative systems.

On Formal Methods and Software Engineering The FM community focuses on mathematics-based techniques for the specification, development, and (manual or automated) verification of software and hardware systems [52]. In contrast, SE mainly focus on generating finite-state machines or graph models. In [22], for example, a communicating finite state machine is generated from a log of system executions enhanced with time vectors. Although model checking facilities over the models are available via the McScM tool, an automatic way to compose the models generated from different logs is not provided. In [73], message sequence graphs are mined from logs of distributed systems; these models are used for program comprehension, since they provides an higher-level view of the system behavior and no verification technique is mentioned to analyse the obtained models. The authors of [118] exploit an idea close to our work, presenting an algorithm to construct the overall model of a system by composing models of its components. The main difference with respect to our work is that they infer a model by analisying a list of log messages, knowing a priori the architecture dependencies among the components of the distributed system. The output of this inference process is a FSM. The work focuses on the scalability problem of large systems, while no mention to possible verification techniques is given. Other techniques, like the one proposed in [51], focus on building decision trees from message logs to detect possible failures in the system.

On FM side a closely related line of research with this work concerns automata learning; the aim is to construct an automaton by providing inputs to a system and observing the corresponding outputs [129]. In this context, there are two types of learning: active and passive. In the former one (see, e.g., [4, 64, 10]), experiments are carried out over the system, while the latter one (see, e.g., [91, 134, 60, 59]) is based on generated runs (i.e., logs). Our approach differs from the ones proposed by the automata learning community for the input and the output of the process: we consider as input logs instead of automaton or traces, and we produce as output a process algebraic specification (in particular, a customised mCRL2 specification) instead of automata (FSM, state diagrams, I/O automata, etc.). Tailoring of the automata learning techniques to process algebraic specification mining certainly deserves an in-depth investigation.

Chapter 5

Concluding Remarks

This thesis proposes two approaches, namely the *top-down* and *bottom-up approach*, that, by combining formal methods and business process management techniques, aim at enabling verification capabilities on distributed activities taking advantage of the well-studied verification capabilities of FM and the powerful process mining techniques to discover models automatically.

As demonstrated in Chapters 3 and 4, a combination of Formal Methods (FM) and Business Process Management (BPM) techniques proves effective in enabling formal verification of existing models, facilitating the verification of properties—such as privacy-related ones. A top-down approach allows to support and verify activities at a design level. Additionally, employing a bottom-up approach aids in verifying operational processes to identify and rectify misconfigurations or unexpected events occurring within distributed activities, particularly in cases involving intercommunications between such activities.

On the one hand, we deal with the challenge of verifying already existing models, like BPMN or PE-BPMN models, to support the design phase of distributed systems by allowing correctness verification techniques on the models' behaviour or the implementation of specific features on them. In particular, the *top-down approach* in Section 3 proposes a method for verifying BPMN collaborations enhanced with privacy-enhancing technology measures. The proposed method detects situations where privacy-enhancing technologies are misused, which may lead to unauthorised access to private data. In addition, the method supports the verification of other properties, such as the reconstruction of a shared secret, the parallel execution of multi-party computations, and deadlock freedom to provide proof of design correctness. The method is based on a formalisation of PE-BPMN collaborations in terms of $mCRL2$ specifications, where PETs and properties to check are embedded as part of the specification. The proposed approach has been implemented as a tool available as a stand-alone application and a plug-in in the Pleak toolset for business process privacy analysis.

On the other hand, we deal with the problem of verifying properties of already up-and-running distributed activities and with the difficulty of designing models that allow us to detect their real behaviour and execute verification of properties to identify unexpected anomalies or misbehaviours. The *bottom-up* approach in Section 4 tackles this problem by proposing a methodology, called PALM, which is a technique to automatically generate the model of a system behaviour from a set of observations. In particular, taking as input event logs of a distributed system, it can produce, in a compositional way, a formal specification of the system's overall behaviour in the $nCRL2$ language, i.e. a fragment of the $mCRL2$ language. This enables us to take advantage of the $mCRL2$ toolset for formal verification, enabling the detection of issues that may arise in a distributed scenario where multiple organisations interact to reach a common goal. The methodology is validated empirically using custom-made and real event logs and through a complete demonstration that validates the semantic correctness of the approach by providing a formal proof based on operational correspondence between the languages used in the methodology, i.e. block structure, $nCRL2$ and $mCRL2$.

Summing up, both approaches use $mCRL2$ as modelling language to verify distributed activities in two typical phases: designing new distributed activities or validating existing ones. In the context of analysing the applicability of top-down and bottom-up approaches in the verification of distributed activities, it becomes evident that the bottom-up can suffer from performance issues due to the size of logs, while top-down approaches usually have to deal with models less complex, especially when are directly designed by the modeller. Conversely, the bottom-up approach is necessary to capture the actual behaviour of individual components and the system as a whole, as the modelled representation may only sometimes align with reality. Throughout the investigation, several lessons were learned:

- The analysis process can be time-consuming for large logs or models, which may discourage its usage.

- Embedded properties are essential to verify abnormal behaviour or behaviour that deviates from the expected model (e.g. privacy-enhancing technologies properties). This allows for efficiently identifying violations and deviations from the desired properties.

- Providing counterexamples to identify where a particular property is violated proves valuable, as it enables immediate problem identification and the implementation of necessary corrective measures.

These insights emphasise the importance of considering the trade-offs between top-down and bottom-up approaches in verifying distributed activities. The top-down approach offers advantages in managing log size and complexity while dealing with design properties, while the bottom-up approach provides a more accurate representation of the actual system behaviour. Moreover, combining the top-down and bottom-up approaches provides a more robust framework for verifying distributed activities, allowing for a thorough examination of both the system-level design and the individual components' functionality and compliance. This holistic approach could enhance the accuracy and effectiveness of the verification process, leading to improved system performance and reliability in a wide range of domains and applications.

5.1 Future Work

Combining FM and BPM can effectively enable verification for distributed activities. Still, many aspects can be improved for both approaches.

We aim to tackle the current limitations of the top-down approach. First, we intend to improve the transformation of loops that are now limited to mimic the behaviour of while loops in programming languages by discriminating between the forwards and roll-back part of the loop. Moreover, currently, the generated specification allows handling only single-instance models and does not support dynamic instantiation of processes, meaning that a specific workflow related to a participant can be instantiated just through the start event once. At the same time, it is useful to manage multi-instance and dynamic instantiation of processes, especially when modelling scenarios involving services components (e.g., in SaaS applications) that can be instantiated at any time, depending on the needs. Considering possible extensions, the top-down approach could capture other privacy-related properties that can be defined on PE-BPMN models, such as *anonymity*, i.e., the ability of a participant to make an element non-identifiable to other participants, or *unlinkability*, i.e., the impossibility of a participant to understand if two data objects are related. Moreover, the proposed transformation from PE-BPMN collaborations is currently restricted to collaborations where each party's process is block-structured. Since the class of block-structured BPMN process models is relatively expressive [100], lifting this restriction would be desirable. A challenge here is how to lift this restriction while still taking advantage of the compositionality of the process algebraic approach to obtain manageable formal specifications. Finally, a potential area of interest for further investigation is the potential inclusion of self-loops on the end state in order to enhance the existing workaround employed for verifying deadlock freedom. By introducing self-loops, it may be possible to avoid the reachability checking while using a specific deadlock formula available in mCRL2.

While the main challenges of the top-down approaches focus on formalising properties and verifying them, on the bottom-up approach, we aim first to investigate how to improve our mining algorithm's capability of detecting the choice between performing an action and skipping it to improve the generated model. It could be interesting to extend the target language of the PALM methodology with data and time features to develop richer specifications and extend the verification capabilities to the data dimension as done in the top-down approach. Of course, this kind of information must be present in the input logs. In particular, since many distributed systems also implement privacy properties, it could be interesting to extend the approach and make it able to discover PETs or other privacy-related features automatically.

It is also important to extend our validation experiments, including equivalence checking to other process mining algorithms to constantly measure the capability of the proposed one and improve the replication of the validation experiments, which currently is only partially supported by the application, by integrating the TDKE Benchmark and the ProM plug-ins to generate the BPMN models and consequently the Petri Nets and Reachability graphs. The validation can be extended to consider further measurements, like alignments that provide a fine-granular approach to detecting deviations on the level of individual events and task execution [31].

Appendix A

Proofs to Establish Semantic Correctness

A.1 Operational Correspondence Between $mCRL2$ and $nCRL2$ and Viceversa

Definition A.1.1 ($mCRL2$ Process Specification Semantics)**.** *The semantics of a process specification $\langle q, E\rangle$ is a LTS $(S, L, \rightarrow)$ as follows:*

- *The set of states S (ranged over by s) contains process specifications and one special termination state, denoted by $\checkmark$.*
- *The set of labels L (ranged over by α) is generated by the following grammar:*

$$\alpha ::= \tau \;\mid\; a \;\mid\; \alpha_1 | \alpha_2$$

- *The transitions relation $\rightarrow\ \subseteq S \times L \times S$ is inductively defined by the following operational rules in table 13. Also, in this case, there is no need for a symmetric version of rules since $mCRL2$ identifies process expressions up to commutativity and associativity of choice and parallel operator. We will write $s_1 \xrightarrow{\alpha} s_2$ to indicate that $(s_1, \alpha, s_2) \in\rightarrow$ and, to improve the readability of the operational rules, we omit the set of process equations when they play no role in a rule, e.g. writing $\langle q, E\rangle \xrightarrow{\alpha} \langle q', E\rangle$ as $q \xrightarrow{\alpha} q'$, or $\langle q, E\rangle \xrightarrow{\alpha} \checkmark$ as $\xrightarrow{\alpha} \checkmark$*

- s_0 *is the initial state.*

Following we report part of the $mCRL2$ syntax and semantics useful to understand the proofs and the complete demonstration of the Propositions 1 and 2 validity.

ACT	$\frac{}{\alpha \xrightarrow{[\![\alpha]\!]}_u \checkmark}$	CH$_1$	$\frac{p \xrightarrow{\alpha}_u \checkmark}{p{+}q \xrightarrow{\alpha}_u \checkmark}$
CH$_2$	$\frac{p \xrightarrow{\alpha}_u p'}{p{+}q \xrightarrow{\alpha}_u p'}$	SQ$_1$	$\frac{p \xrightarrow{\alpha}_u \checkmark}{p.q \xrightarrow{\alpha}_u t_u{>>}q}$
SQ$_2$	$\frac{p \xrightarrow{\alpha}_u p'}{p.q \xrightarrow{\alpha}_u p'.q}$	REC$_1$	$\frac{Q{=}q{\in}E \quad q \xrightarrow{\alpha}_u \checkmark}{\langle Q{,}E\rangle \xrightarrow{\alpha}_u \checkmark}$
REC$_2$	$\frac{Q{=}q{\in}E \quad q \xrightarrow{\alpha}_u q'}{\langle Q{,}E\rangle \xrightarrow{\alpha}_u q'}$	PAR$_1$	$\frac{p \xrightarrow{\alpha}_u \checkmark, q \rightsquigarrow_u}{p\vert\vert q \xrightarrow{\alpha}_u t{>>}_u q}$
PAR$_2$	$\frac{p \xrightarrow{\alpha}_u p', q \rightsquigarrow_u}{p\vert\vert q \xrightarrow{\alpha}_u p'\vert\vert t{>>}_u q}$	PARC$_1$	$\frac{p \xrightarrow{\alpha}_u \checkmark \; q \xrightarrow{\bar{\alpha}}_u \checkmark}{p\vert\vert q \xrightarrow{\alpha\vert\bar{\alpha}}_u \checkmark}$
PARC$_2$	$\frac{p \xrightarrow{\alpha}_u p' \; q \xrightarrow{\bar{\alpha}}_u \checkmark}{p\vert\vert q \xrightarrow{\alpha\vert\bar{\alpha}}_u p'}$	PARC$^{\text{n}}{}_4$	$\frac{p \xrightarrow{\alpha}_u p' \; q \xrightarrow{\bar{\alpha}}_u q'}{p\vert\vert q \xrightarrow{\alpha\vert\bar{\alpha}}_u p'\vert\vert q'}$
BI$_1$	$\frac{q \xrightarrow{\alpha}_u \checkmark}{t{>>}q \xrightarrow{\alpha}_u \checkmark} \; u > [\![t]\!]$	BI$_2$	$\frac{q \xrightarrow{\alpha}_u q'}{t{>>}q \xrightarrow{\alpha}_u q'} \; u > [\![t]\!]$
BI$_3$	$\frac{p \rightsquigarrow_u}{t{>>}p \rightsquigarrow_u}$	BI$_4$	$\frac{}{t{>>}p \rightsquigarrow_u} \; u < [\![t]\!]$
ALL$_1$	$\frac{p \xrightarrow{\alpha}_u \checkmark}{\nabla_V(p) \xrightarrow{\alpha}_u \checkmark} \alpha \in V \cup \{\tau\}$	ALL$_2$	$\frac{p \xrightarrow{\alpha}_u p'}{\nabla_V(p) \xrightarrow{\alpha}_u \nabla_V(p')}$
COM$_1$	$\frac{p \xrightarrow{\alpha}_u \checkmark}{\Gamma_C(p) \xrightarrow{\gamma_C(\alpha)}_u \checkmark}$	COM$_2$	$\frac{p \xrightarrow{\alpha}_u p'}{\Gamma_C(p) \xrightarrow{\gamma_C(\alpha)}_u \Gamma_C(p')}$
HD$_1$	$\frac{p \xrightarrow{\alpha}_u \checkmark}{\tau_I(p) \xrightarrow{\theta_I(\alpha)}_u \checkmark}$	HD$_2$	$\frac{p \xrightarrow{\alpha}_u p'}{\tau_I(p) \xrightarrow{\theta_I(\alpha)}_u \tau_I(p')}$

Table 13: SOS $mCRL2$

For the complete list of structural operational semantics rules of $mCRL2$ refer to [56]. Following we show the proofs related to Propositions 1 and 2.

Proof. The proof for Proposition 1 is done by induction on the depth of inference. First of all we show that the proof is satisfied for the base case, i.e. for the axioms ACT^n and ACT. By case analysis on the value of α:

- $\alpha = \tau : \tau \xrightarrow{\tau} \checkmark$, then $\exists\ u$ s.t. $\tau \xrightarrow{\tau}_u$ and as expected $\phi(\checkmark)$ = $\checkmark$.
- $\alpha = a : a \xrightarrow{a} \checkmark$, then $\exists\ u$ s.t. $a \xrightarrow{a}_u \checkmark$ and as expected $\phi(\checkmark)$ = $\checkmark$
- $\alpha = \alpha_1|\alpha_2 : \alpha_1|\alpha_2 \xrightarrow{\alpha_1|\alpha_2} \checkmark$, then $\exists\ u$ s.t. $\alpha_1|\alpha_2 \xrightarrow{\alpha_1|\alpha_2}_u \checkmark$ and as expected $\phi(\checkmark)$ = $\checkmark$

Then Proposition 1 holds for the base case.
Since for an inference of length n the proposition holds by induction, we need to prove that it holds for $n + 1$. The proof is done by case analysis on each operator of the $nCRL2$ syntax.

- ⋆ $\mathbf{CH^n}_1$: $p + q \xrightarrow{\alpha} \checkmark$ based on the premises this is valid if $p \xrightarrow{\alpha} \checkmark$, since this is done on a shorter derivation we can apply the inductive hypothesis and we obtain that $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u \checkmark$ and applying CH$_1$ we obtain $p + q \xrightarrow{\alpha}_u \checkmark$ and $\phi(\checkmark) = \checkmark$ as expected.
- ⋆ $\mathbf{CH^n}_2$: $p+q \xrightarrow{\alpha} p'$ this is valid if $p \xrightarrow{\alpha} p'$, by applying the inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u p'$ and applying CH$_2$ we obtain $p+q \xrightarrow{\alpha}_u p'$ and $\phi(p') = p'$ as expected
- ⋆ $\mathbf{SQ^n}_1$: $p.q \xrightarrow{\alpha} q$ given the premises this is valid if $p \xrightarrow{\alpha} \checkmark$, since this is done on a shorter derivation we can apply the inductive hypothesis and we obtain that $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u \checkmark$ and applying SQ$_1$ we obtain $p.q \xrightarrow{\alpha}_u t_u >> q$ and $\phi(t_u >> q) = q$ as we expected.
- ⋆ $\mathbf{SQ^n}_2$: $p.q \xrightarrow{\alpha} p'.q$ this is valid if $p \xrightarrow{\alpha} p'$, by applying the inductive inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u p'$ and applying SQ$_2$ we obtain $p.q \xrightarrow{\alpha}_u p'.q$ and $\phi(p'.q) = p'.q$ as expected
- ⋆ $\mathbf{REC^n}_1$: $\langle P, E\rangle \xrightarrow{\alpha} \checkmark$ this is valid if $P = p \in E \quad p \xrightarrow{\alpha} \checkmark$ by inductive hypothesis $\exists\ u$ s.t. $q \xrightarrow{\alpha}_u \checkmark$ and applying REC$_1$ we obtain $P \xrightarrow{\alpha}_u \checkmark$ and $\phi(\checkmark) = \checkmark$ as expected
- ⋆ $\mathbf{REC^n}_2$: $\langle P, E\rangle \xrightarrow{\alpha} \langle p', E\rangle$ this is valid if $P = p \in E \quad p \xrightarrow{\alpha} p'$ by inductive hypothesis $\exists\ u$ s.t. $q \xrightarrow{\alpha}_u q'$ and applying REC$_2$ $P \xrightarrow{\alpha}_u q'$ and $\phi(q') = p'$ as expected

- $\star$ **PAR$^{\mathbf{n}}$**$_1$: $p||q \xrightarrow{\alpha} q$ this is valid if $p \xrightarrow{\alpha} \checkmark$ by inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u \checkmark$ and applying PAR$_1$ we obtain $p||q \xrightarrow{\alpha}_u t >>_u q$ and $\phi(t >>_u q) = q$ as expected

- $\star$ **PAR$^{\mathbf{n}}$**$_2$: $p||q \xrightarrow{\alpha} p'||q$ this is valid if $p \xrightarrow{\alpha} p'$ by inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u p'$ and applying PAR$_2$ we obtain $p||q \xrightarrow{\alpha}_u p'||t >>_u q$ and $\phi(p'||t >>_u q) = p'||q$ as expected

- $\star$ **PARC$^{\mathbf{n}}$**$_1$: $p||q \xrightarrow{\alpha|\beta} \checkmark$ this is valid if $p \xrightarrow{\alpha} \checkmark q \xrightarrow{\beta} \checkmark$ by inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u \checkmark q \xrightarrow{\beta}_u \checkmark$ and applying PARC$_1$ we obtain $p||q \xrightarrow{\alpha|\beta}_u \checkmark$ and $\phi(\checkmark) = \checkmark$ as expected

- $\star$ **PARC$^{\mathbf{n}}$**$_2$: $p||q \xrightarrow{\alpha|\beta} p'$ this is valid if $p \xrightarrow{\alpha} p'$ and $q \xrightarrow{\beta} \checkmark$ by inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u p' q \xrightarrow{\beta}_u \checkmark$ and applying PARC$_2$ we obtain $p||q \xrightarrow{\alpha|\beta}_u p'$ and $\phi(p') = p'$ as expected

- $\star$ **PARC$^{\mathbf{n}}$**$_4$: $p||q \xrightarrow{\alpha|\beta}_u p'||q'$ this is valid if $p \xrightarrow{\alpha} p'$ and $q \xrightarrow{\beta} q'$ by inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u p'\ q \xrightarrow{\beta}_u q'$ and applying PARC$_4$ we obtain $p||q \xrightarrow{\alpha|\beta} p'||q'$ and $\phi(p'||q') = p'||q'$ as expected

- $\star$ **ALL$^{\mathbf{n}}$**$_1$: $allow(V, p) \xrightarrow{\alpha} \checkmark$ this is valid if $p \xrightarrow{\alpha} \checkmark$, by inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u \checkmark$ and applying ALL$_1$ we obtain $\nabla_V(p) \xrightarrow{\alpha}_u \checkmark$ and $\phi(\checkmark) = \checkmark$ as expected

- $\star$ **ALL$^{\mathbf{n}}$**$_2$: $allow(V, p) \xrightarrow{\alpha} allow(V, p')$ this is valid if $p \xrightarrow{\alpha} p'$, by inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha}_u p'$ and applying ALL$_2$ we obtain $\nabla_V(p) \xrightarrow{\alpha}_u p'$ and $\phi(p') = p'$ as expected

- $\star$ **COM$^{\mathbf{n}}$**$_1$, **COM$^{\mathbf{n}}$**$_2$, **HD$^{\mathbf{n}}$**$_1$ and **HD$^{\mathbf{n}}$**$_2$, these cases are omitted because they are similar to ALL$^{\text{n}}{}_1$ and ALL$^{\text{n}}{}_2$

□

Proof. The proof for Proposition 2, like the one for Proposition 1, is done by induction on the depth of inference.
First of all we show that the proof is satisfied for the base case, i.e. for the axioms ACT and ACT$^{\text{n}}$. By case analysis on the value of α:

- $\alpha = \tau : \tau \xrightarrow{\tau}_u \checkmark$, then $\tau \xrightarrow{\tau}$ and as expected $\phi(\checkmark) = \checkmark$.

- $\alpha = a : a \xrightarrow{a}_u \checkmark$, then $a \xrightarrow{a} \checkmark$ and as expected $\phi(\checkmark) = \checkmark$

- $\alpha = \alpha_1|\alpha_2 : \alpha_1|\alpha_2 \xrightarrow{\alpha_1|\alpha_2}_u \checkmark$, then $\alpha_1|\alpha_2 \xrightarrow{\alpha_1|\alpha_2} \checkmark$ and as expected $\phi(\checkmark) = \checkmark$

Then Proposition 2 holds for the base case. Since for an inference of length n the proposition holds by induction, we need to prove they holds for $n + 1$. The proof is done by case analysis on each operator of the $mCRL2$ syntax such that $p \in$ mCRL2.

⋆ **CH**$_1$: $p + q \xrightarrow{\alpha}_u \checkmark$ based on the premises this is valid if $p \xrightarrow{\alpha}_u \checkmark$, since this is done on a shorter derivation we can apply the inductive hypothesis and we obtain that $p \xrightarrow{\alpha} \checkmark$ and applying CH$^{\mathrm{n}}{}_1$ we obtain $p + q \xrightarrow{\alpha} \checkmark$ and $\phi(\checkmark) = \checkmark$ as expected.

⋆ **CH**$_2$: $p+q \xrightarrow{\alpha}_u p'$ this is valid if $p \xrightarrow{\alpha}_u p'$, by applying the inductive hypothesis $p \xrightarrow{\alpha} p'$ and applying CH$^{\mathrm{n}}{}_2$ we obtain $p + q \xrightarrow{\alpha} p'$ and $\phi(p') = p'$ as expected

⋆ **SQ**$_1$: $p.q \xrightarrow{\alpha}_u t_u >> q$ given the premises this is valid if $p \xrightarrow{\alpha}_u \checkmark$, since this is done on a shorter derivation we can apply the inductive hypothesis and we obtain that $p \xrightarrow{\alpha} \checkmark$ and applying SQ$^{\mathrm{n}}{}_1$ we obtain $p.q \xrightarrow{\alpha}_u q$ and $\phi(t_u >> q) = q$ as we expected.

⋆ **SQ**$_2$: $p.q \xrightarrow{\alpha}_u p'.q$ this is valid if $p \xrightarrow{\alpha}_u p'$, by applying the inductive inductive hypothesis $p \xrightarrow{\alpha} p'$ and applying SQ$^{\mathrm{n}}{}_2$ we obtain $p.q \xrightarrow{\alpha} p'.q$ and $\phi(p'.q) = p'.q$ as expected

⋆ **REC**$_1$: $\langle Q, E\rangle \xrightarrow{\alpha}_u \checkmark$ this is valid if $Q = q \in E$ and $q \xrightarrow{\alpha}_u \checkmark$ by inductive hypothesis $q \xrightarrow{\alpha} \checkmark$ with $Q = q \in E$ and applying REC$^{\mathrm{n}}{}_1$ we obtain $\langle Q, E\rangle \xrightarrow{\alpha} \checkmark$ and $\phi(\checkmark) = \checkmark$ as expected

⋆ **REC**$_2$: $\langle Q, E\rangle \xrightarrow{\alpha}_u q'$ this is valid if $Q = q \in E$ and $q \xrightarrow{\alpha}_u q'$ by inductive hypothesis $q \xrightarrow{\alpha} q'$ with $Q = q \in E$ and applying REC$^{\mathrm{n}}{}_2$ $\langle Q, E\rangle \xrightarrow{\alpha} q'$ and $\phi(q') = q'$ as expected

⋆ **PAR**$_1$: $p||q \xrightarrow{\alpha}_u t >>_u q$ this is valid if $p \xrightarrow{\alpha}_u \checkmark$ by inductive hypothesis $p \xrightarrow{\alpha} \checkmark$ and applying PAR$^{\mathrm{n}}{}_1$ we obtain $p||q \xrightarrow{\alpha} q$ and $\phi(t >>_u q) = q$ as expected

- $\star$ **PAR**$_2$: $p||q \xrightarrow{\alpha}_u p'||t >>_u q$ this is valid if $p \xrightarrow{\alpha}_u p'$ by inductive hypothesis $p \xrightarrow{\alpha} p'$ and applying $\text{PAR}^{\text{n}}{}_2$ we obtain $p||q \xrightarrow{\alpha} p'$ and $\phi(p'||t >>_u q) = p'||q$ as expected

- $\star$ **PAR**$_3$ and **PAR**$_4$, these cases are omitted because they are symmetric with respect to the cases of rules $\text{PAR}^{\text{n}}{}_1$ and $\text{PAR}^{\text{n}}{}_2$

- $\star$ **PARC**$_1$: $p||q \xrightarrow{\alpha|\beta}_u \checkmark$ this is valid if $p \xrightarrow{\alpha}_u \checkmark$ and $q \xrightarrow{\alpha}_u \checkmark$ by inductive hypothesis $p \xrightarrow{\alpha} \checkmark$ and $q \xrightarrow{\alpha} \checkmark$ and applying $\text{PARC}^{\text{n}}{}_1$ we obtain $p||q \xrightarrow{\alpha|\beta} \checkmark$ and $\phi(\checkmark) = \checkmark$ as expected

- $\star$ **PARC**$_2$: $p||q \xrightarrow{\alpha|\beta}_u p'$ this is valid if $p \xrightarrow{\alpha}_u p'$ and $q \xrightarrow{\alpha}_u \checkmark$ by inductive hypothesis $p \xrightarrow{\alpha} p'$ and $q \xrightarrow{\alpha} \checkmark$ and applying $\text{PARC}^{\text{n}}{}_2$ we obtain $p||q \xrightarrow{\alpha|\beta} p'$ and $\phi(p') = p'$ as expected

- $\star$ **PARC**$_4$: $p||q \xrightarrow{\alpha|\beta}_u p'||q'$ this is valid if $p \xrightarrow{\alpha}_u p'$ and $q \xrightarrow{\beta} q'$ by inductive hypothesis $\exists\ u$ s.t. $p \xrightarrow{\alpha} p'$ and $q \xrightarrow{\beta} q'$ and applying $\text{PARC}^{\text{n}}{}_4$ we obtain $p||q \xrightarrow{\alpha|\beta} p'||q'$ and $\phi(p'||q') = p'||q'$

- $\star$ **ALL**$_1$: $allow(V, p) \xrightarrow{\alpha}_u \checkmark$ this is valid if $p \xrightarrow{\alpha}_u \checkmark$, by inductive hypothesis $p \xrightarrow{\alpha} \checkmark$ and applying $\text{ALL}^{\text{n}}{}_1$ we obtain $\nabla_V(p) \xrightarrow{\alpha} \checkmark$ and $\phi(\checkmark) = \checkmark$ as expected

- $\star$ **ALL**$_2$: $allow(V, p) \xrightarrow{\alpha}_u allow(V, p')$ this is valid if $p \xrightarrow{\alpha}_u p'$, by inductive hypothesis $p \xrightarrow{\alpha} p'$ and applying $\text{ALL}^{\text{n}}{}_2$ we obtain $\nabla_V(p) \xrightarrow{\alpha} p'$ and $\phi(p') = p'$ as expected

- $\star$ **COM**$_1$, **COM**$_2$, **HD**$_1$ and **HD**$_2$, these case are omitted because they are similar to ALL$_1$ and ALL$_2$

□

A.2 Operational Correspondence Between B and $nCRL2$ and Viceversa

Proof. The proof for Proposition 3 is done by induction on the depth of the inference. First of all we show that the proof is satisfied for the base case, i.e. for the axiom op_1. Base case:

- $\star$ **op**$_1$ $a \xrightarrow{a} \checkmark$, then $\mathcal{T}(a) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and $\mathcal{T}(\checkmark) = \mathcal{T}(\checkmark)$

- $\star$ **Lop**$_1$ $L\{B\} \xrightarrow{\tau} \checkmark$ then $\text{REC}^{\text{N}}{}_1 \dfrac{\text{CH}^{\text{N}}{}_3 \dfrac{\mathcal{T}(\tau) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)}{\mathcal{T}(B).P + \mathcal{T}(\tau) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)}}{P \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)}$, and $\mathcal{T}(\checkmark) = \mathcal{T}(\checkmark)$

Then Proposition 3 holds for the base case. Since for an inference of length n the proposition holds by induction, we need to prove that it hols also for $n+1$. The proof is done by case analysis on each operator of the block structure.

- $\star$ **Sop**$_1$: $S\{B_1, ..., B_n\} \xrightarrow{a} S\{B_1', ..., B_n\}$ based on the premises this is valid if $B_1 \xrightarrow{a} B_1'$ with $n \geq 1$, since this is done on a shorter derivation we can apply the inductive hypothesis and we obtain $\mathcal{T}(B_1) \overset{a}{\rightsquigarrow} \mathcal{T}(B_1'')$ and applying SQ_2^n we obtain: $\mathcal{T}(S\{B_1 \dots B_n\}) =$ $\mathcal{T}(B_1).\mathcal{T}(B_2) \dots \mathcal{T}(B_n) \overset{a}{\rightsquigarrow} \mathcal{T}(B_1'').\mathcal{T}(B_2) \dots \mathcal{T}(B_n)$ and $\mathcal{T}(B_1').\mathcal{T}(B_2) \dots \mathcal{T}(B_n) = \mathcal{T}(B_1'').\mathcal{T}(B_2) \dots \mathcal{T}(B_n)$

- $\star$ **Sop**$_2$: $S\{B_1, ..., B_n\} \xrightarrow{a} S\{B_2, ..., B_n\}$ based on the premises this is valid if $B_1 \xrightarrow{a} \checkmark$ with $n > 1$, since this is done on a shorter derivation we can apply the inductive hypothesis and we obtain $\mathcal{T}(B_1) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and applying SQ_1^n we obtain: $\mathcal{T}(S\{B_1 \dots B_n\}) =$ $\mathcal{T}(B_1).\mathcal{T}(B_2) \dots \mathcal{T}(B_n) \overset{a}{\rightsquigarrow} \mathcal{T}(B_2).\mathcal{T}(B_3) \dots \mathcal{T}(B_n)$ and $\mathcal{T}(B_2).\mathcal{T}(B_3) \dots \mathcal{T}(B_n) = \mathcal{T}(B_2).trans(B_3) \dots \mathcal{T}(B_n)$

- $\star$ **Sop**$_3$: $S\{B\} \xrightarrow{a} \checkmark$ based on the premises this is valid if $B \xrightarrow{a} \checkmark$, since this is done on a shorter derivation we can apply the inductive hypothesis and we obtain $\mathcal{T}(B) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and applying SQ_1^n we obtain $\mathcal{T}(B) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and $\mathcal{T}(\checkmark) = \mathcal{T}(\checkmark)$

- $\star$ **Pop**$_1$: $P\{B_1, \dots, B_j, \dots, B_n\} \xrightarrow{a} P\{B_1, \dots, B_j', \dots, B_n\}$ based on the premises this is valid if $B_j \xrightarrow{a} B_j'$ with $1 \leq j \leq n$, by applying the inductive hypothesis $\mathcal{T}(B_j) \overset{a}{\rightsquigarrow} \mathcal{T}(B_j'')$ and applying PAR_2^n we obtain: $\mathcal{T}(P\{B_1 || \dots || B_j || \dots \ || B_n\}) =$ $\mathcal{T}(B_1) || \dots || \mathcal{T}(B_j) || \dots || \mathcal{T}(B_n) \overset{a}{\rightsquigarrow} \mathcal{T}(B_1) || \dots || \mathcal{T}(B_j'') || \dots || \mathcal{T}(B_n)$ and $\mathcal{T}(B_1) || \dots || \mathcal{T}(B_j') || \dots || \mathcal{T}(B_n) = \mathcal{T}(B_1) || \dots || \mathcal{T}(B_j'') || \dots || \mathcal{T}(B_n)$

- $\star$ **Pop**$_2$: $P\{B_1, \dots, B_j, \dots, B_n\} \xrightarrow{a} P\{B_1, \dots, B_n\}$ based on the premises this is valid if $B_j \xrightarrow{a} \checkmark$ with $n > 1$, by applying the inductive hypothesis $\mathcal{T}(B_j) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and applying PAR_1^n we obtain $\mathcal{T}(P\{B_1|| \dots ||B_j|| \dots ||B_n\}) =$ $\mathcal{T}(B_1)|| \dots ||\mathcal{T}(B_j)|| \dots ||\mathcal{T}(B_n) \overset{a}{\rightsquigarrow} \mathcal{T}(B_1)|| \dots ||\mathcal{T}(B_n)\backslash\{B_j\}$ and $\mathcal{T}(B_1)|| \dots ||\mathcal{T}(B_n)\backslash\{B_j\} = \mathcal{T}(B_1)|| \dots ||\mathcal{T}(B_n)\backslash\{B_j\}$

- ⋆ **Pop**$_3$ $P\{B_1, \dots, B_j, \dots, B_n\} \xrightarrow{a} \checkmark$ based on the premises this is valid if $B_j \xrightarrow{a} \checkmark$, by applying the inductive hypothesis $\mathcal{T}(B_j) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and applying PAR_1^n we obtain $\mathcal{T}(P\{B\}) = \mathcal{T}(B) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and $\mathcal{T}(\checkmark) = \mathcal{T}(\checkmark)$

- ⋆ **Cop**$_1$ $C\{B_1, \dots, B_j, \dots, B_n\} \xrightarrow{a} B'_j$ based on the premises this is valid if $B_j \xrightarrow{a} B'_j$, by applying the inductive hypothesis $\mathcal{T}(B_j) \overset{a}{\rightsquigarrow} \mathcal{T}(B''_j)$ and applying $CH^{\text{n}}{}_2$ we obtain $\mathcal{T}(C\{B_1, \dots, B_j, \dots, B_n\}) = \mathcal{T}(B_1) + \dots + \mathcal{T}(B_j) + \dots + \mathcal{T}(B_n) \overset{a}{\rightsquigarrow} \mathcal{T}(B''_j)$ and $\mathcal{T}(B'_j) = \mathcal{T}(B''_j)$

- ⋆ **Cop**$_2$ $C\{B_1, \dots, B_j, \dots, B_n\} \xrightarrow{a} \checkmark$ based on the premises this is valid if $B_j \xrightarrow{a} \checkmark'$, by applying the inductive hypothesis $\mathcal{T}(B_j) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and applying $CH^{\text{n}}{}_1$ we obtain $\mathcal{T}(C\{B_1, \dots, B_j, \dots, B_n\}) = \mathcal{T}(B_1) + \dots + \mathcal{T}(B_j) + \dots + \mathcal{T}(B_n) \overset{a}{\rightsquigarrow} \mathcal{T}(\checkmark)$ and $\mathcal{T}(\checkmark) = \mathcal{T}(\checkmark)$

- ⋆ **Lop**$_2$: $L\{B\} \xrightarrow{a} S\{B', L\{B\}\}$ based on the premises this is valid if $B \xrightarrow{a} B'$ by applying the inductive hypothesis $\mathcal{T}(B) \overset{a}{\rightsquigarrow} \mathcal{T}(B'')$ and applying a sequence of $nCRL2$ semantic rules we obtain

$$\text{REC}^{\text{N}}{}_2\ \dfrac{\text{CH}^{\text{N}}{}_2\ \dfrac{\text{SQ}^{\text{N}}{}_2\ \dfrac{\mathcal{T}(B) \overset{a}{\rightsquigarrow} \mathcal{T}(B'')}{\mathcal{T}(B).P \overset{a}{\rightsquigarrow} \mathcal{T}(B'').P}}{\mathcal{T}(B).P + \mathcal{T}(\tau) \overset{a}{\rightsquigarrow} \mathcal{T}(B'').P}}{P \overset{a}{\rightsquigarrow} \mathcal{T}(B'').P}$$

, and $\mathcal{T}(S\{B', L\{B\}\}) = \mathcal{T}(B').\mathcal{T}(L\{B\}) = \mathcal{T}(B'').P$

□

Proof. The proof for Proposition 4 should be done by induction on the depth of the inference exploiting the case analysis for each operator. Since for each case, the proof is symmetric to the proof of Proposition 3 we omit it. □

Appendix B

PALM Details

B.1 Mining Tool-independent Specification

Following, we report part of the code implementing the PALM methodology.

The mining algorithm is computed by the SchimmAlgorithm object while it is instantiated. Every method in the function represents a step of the algorithm except for *unifyAllBlockStructure* that aggregates two steps that are **4th step - model for each cluster** and **5th step - unify all block structures**.

```java
public SchimmAlgorithm(EventLog log) {
 //Set containing all the loops found in this eventlog
 this.loops = new HashSet<LoopSet >();
 this.log = log;
 //1st - search for loops
 searchForLoops();
 //2nd step - creation of clusters
 Set<Trace> cluster = creationOfClusters();
 //3rd step - identification and removal of pseudo-dependencies
 cluster = identificationAndRemovalOfPseudoDependencies(cluster);
 //4th step - model for each cluster && 5th step unify all block structure
 BlockStructure blockStructure = unifyAllBlockStructure(cluster)
 //6th step - restructuring the model
 blockStructure = restructuringTheModel(blockStructure);
 //7th step - replacing loop references
 this.modelToTransf = replacingLoopReference(blockStructure);
}
```

Listing B.1: SchimmAlgorithm class constructor

The first step executed in the computation is the **search for loops** method. This method detects all the loops in the event log, substituting them with their process reference. For each detected loop l={$t = \langle e_1 ... e_i ... e_j ... e_n \rangle$ s.t. $e_j > e_i \Rightarrow l = e_i ... e_j$} and for each instance of l in t we need to substitute the reference to the process l in t updating the happened-before relation of t in the following way $e_{i-1} > e_i$ and $e_j > e_{j+1}$, if there is more than one instance of l next to each other, just one reference will be inserted.

```java
private void searchForLoops() {
 for (Trace t : log) {
  for (int i = 0; i < t.length(); i++) {
   Event e = t.getEvent(i);
   Set<Event> incomingEventsofE = t.getPreEventHB(e);
   Trace tmp = t.getSubTraceFrom(i, t.length());
    for (int j = 1; j < tmp.length(); j++) {
     Event ei = tmp.getEvent(j);
     if (incomingEventsofE.contains(ei)) {
     // counts the number of loop's occurrences inside the trace
      int occ = 0;
      //If exists, retrieve a loop already in loops
      LoopSet loopSet = retriveLoop(e, ei);
      Trace loopTrace;
      int lenght = 0;
      do {
       //The trace that is a loop
       loopTrace = t.getSubTrace(e, ei);
       lenght = loopTrace.length();
       //Set the happened-before relation of the trace in the loop
       loopTrace.setHBrel(t.getSubHBrel(loopTrace));
       loopSet.addLoop(loopTrace);
       loops.add(loopSet);
       // Remove an instance of the loop trace
       int start_index = t.removeSubTrace(loopTrace);
       occ += 1;
       if (!(((start_index - 1) < 0) && (start_index + 1) >= t.length())) {
        if (!t.getEvent(start_index - 1).equals(loopSet.getName())){
         t.add(start_index, loopSet.getName());
        if (start_index > 0)
         t.addPreHBRelation(t.getEvent(start_index - 1),
              loopSet.getName());
        if ((start_index + 1) < t.length())
             t.addPostHBRelation(t.getEvent(start_index + 1),
             loopSet.getName());
       } else { if (start_index < t.length())
         t.addPostHBRelation(t.getEvent(start_index),loopSet.getName());
       }} while (!(loopTrace = t.getSubTrace(e, ei)).isEmpty());
       //Sets how many time the loop is repeated in this trace
       loopSet.setRepetition(occ);
       //Update frequency of the loop int the trace
       t.addLoopWithFrequency(loopSet.getName().getName(), lenght, occ);
       break;}}}}}
```

Listing B.2: searchForLoop() method

From line 27 until line 32, we update the happened-before relation of the loop reference as follows: if a reference to this loop does not exist immediately on the left of the trace, then it is added to the trace and updates the happened-before relation with $e_i > e$ and $e > e_j$, otherwise update the post relation of the reference with the element immediately at its right.

In Listing B.3, all the traces with the same alphabet and the same happened-before relation are grouped in one trace. The set of these traces creates the cluster that will be analysed later.

```
private Set<Trace> creationOfClusters() {
 Set<Trace> cluster = new HashSet<Trace>();
 for (Trace t : log) {
  Map<Event, Set<Event>> hbt = t.getHBRel();
  boolean alreadyExist = false;
  for (Trace c : cluster) {
    //Check if a trace in the cluster has the hb-relation equal to the
         trace t
     if (c.equalHB(hbt)) {
        alreadyExist = true;
        break;
      }
    }
   if (!alreadyExist)
     cluster.add(t);
 }
 return cluster;}
```

Listing B.3: Creation of cluster method

Now that the cluster has been computed, we can identify and remove the traces that contain the pseudo-dependencies.

In lines 11-16, we check if the current trace has a relation of precedence s.t. $e_i > e_j$ or $ej > e_i$, then in every trace with the same alphabet this relation should exist; otherwise, the current trace contains a pseudo-dependency and needs to be removed.

In lines 17-21, we check the opposite scenario, that is, if there is not a relation of precedence between e_i and e_j in the current trace, then the same should be the case for all the other traces with the same alphabet in the cluster, the traces found with that relation are removed from the cluster.

```
1 private Set<Trace> identificationAndRemovalOfPseudoDependencie
2  (Set<Trace> cluster){
3 Set<Trace> clusterToRemove = new HashSet<Trace>();
4 for (Trace t1 : cluster) {
5  if (!Sets.difference(cluster, clusterToRemove).contains(t1))
```

```
   continue;
 for (int i = 0; i < t1.length(); i++) {
  Event ei = t1.getEvent(i);
  for (int j = i + 1; j < t1.length(); j++) {
   Event ej = t1.getEvent(j);
   if (t1.containsRelation(ei, ej) || t1.containsRelation(ej, ej)) {
    for (Trace t2 : Sets.difference(cluster, clusterToRemove)) {
     if (!t2.equals(t1) && t2.getAlphabet().equals(t1.getAlphabet()) &&
         !t2.containsRelation(ei, ej) && !t2.containsRelation(ej, ej)) {
      clusterToRemove.add(t1);
      }
    }
   } else if (!t1.containsRelation(ei, ej) && !t1.containsRelation(ej,
        ej)) {
    for (Trace t2 : Sets.difference(cluster, clusterToRemove)) {
     if (!t2.equals(t1) && t2.getAlphabet().equals(t1.getAlphabet()) &&
         (t2.containsRelation(ei, ej) || t2.containsRelation(ej, ej))) {
      clusterToRemove.add(t2);
 }}}}}}
 if (!clusterToRemove.isEmpty())
  cluster.removeAll(clusterToRemove);
  return cluster;
}
```

Listing B.4: Identification and removal of pseudo-dependencies

The first coarse block structure can be constructed when all the pseudo dependencies have been removed. This method contains the 4th and 5th steps of the algorithm. First, compute the model of each cluster and then put them together using the choice operator if needed, i.e. if there is more than one model.

```
1  private BlockStructure unifyAllBlockStructure(Set<Trace> cluster) {
2   BlockStructure[] realFinalPath = new BlockStructure[cluster.size()];
3   int i = 0;
4   for (Trace c : cluster) {
5    //The model of cluster c is computed
6    BlockStructure p = modelForEachCluster(c);
7    realFinalPath[i] = p;
8    i++;
9    }
10   if (realFinalPath.length == 1)
11    return realFinalPath[0];
12   else
13    return new BlockStructure(realFinalPath, Operator.CHOICE);
14  }
```

Listing B.5: Unifiy all block structure method

In line 6 of Listing B.5 the method to construct a block structure from a single cluster (4th step - model for each cluster) is called. Every path obtained from the cluster is represented as a sequence block. All the paths are then combined into a parallel block structure.

```java
private BlockStructure modelForEachCluster(Trace t) {
 List<List<Event>> path = t.computePaths();
 BlockStructure[] toPutInParallel = new BlockStructure[path.size()];
 int i = 0;
 for (List<Event> list : path) {
  BlockStructure[] tmp = new BlockStructure[list.size()];
  for (int j = 0; j < list.size(); j++)
   tmp[j] = new BlockStructure(list.get(j));
   toPutInParallel[i] = new BlockStructure(tmp, Operator.SEQUENCE);
   i++;
  }
 if (toPutInParallel.length < 2)
  return toPutInParallel[0];
 else
  return new BlockStructure(toPutInParallel, Operator.PARALLEL);
}
```

Listing B.6: Model for each cluster

At the end of the unify method, we have a coarse block structure that does not represent the real behaviour yet. In the restructuring method, we have implemented the set of rules that we apply to obtain the final block structure.

```java
private BlockStructure restructuringTheModel(BlockStructure b) {
 BlockStructure newBlock = null;
 if (b.hasEvent())
  return b;
 else if ((b.getOp().equals(Operator.CHOICE) ||
      b.getOp().equals(Operator.PARALLEL) ||
      b.getOp().equals(Operator.SEQUENCE)) && b.size() == 1) {
  // S{B}->B , C{B}->B , P{B}-> B
  newBlock = TransformationRule.removeOperator(b);
 } else if (b.blockWithSamrOperator()) {
  // Op{B1...Op{e1..en}...Bm} -> Op{B1...,e1,...,en,...Bm}
  newBlock = TransformationRule.identity(b);
 } else if ((b.getOp().equals(Operator.PARALLEL) ||
      b.getOp().equals(Operator.CHOICE)) && b.getFirstRowOp() != null &&
      b.getFirstRowOp().equals(Operator.SEQUENCE)) {
  int number = 0;
  // Commutative of parallel and choice operator
  if ((number = howManyBlock(b, LEFT)) != 0)
   newBlock = TransformationRule.mergeSide(b, LEFT, number);
  else if ((number = howManyBlock(b, RIGHT)) != 0)
   newBlock = TransformationRule.mergeSide(b, RIGHT, number);
  }
 if (newBlock == null) {
  newBlock = new BlockStructure(b.getOp());
  for (int i = 0; i < b.size(); i++)
   newBlock.addBlockAtPosition(restructuringTheModel(b.getBlock(i)), i);
 }
 if (!b.equals(newBlock))
  b = restructuringTheModel(newBlock);
 return b;}
```

Listing B.7: Restructuring the model method

At the start of this computation, we substituted the subtraces generating loops with a reference. Now it is time to substitute that reference with the result of the algorithm applied over the subtrace (Listing B.8).

```
private BlockStructure replacingLoopReference(BlockStructure b) {
 if (loops.isEmpty())
  return b;
 Map<Event, BlockStructure> loopNametoLoopBS = new HashMap<Event,
      BlockStructure>();
 for (LoopSet l : loops) {
  SchimmAlgorithm alg = new SchimmAlgorithm(l.getLoop());
  BlockStructure bl = new BlockStructure(Operator.LOOP);
  bl.addBlockAtPosition(alg.getFinalModel(), 0);
  bl.setRepetition(l.getRepetition());
  bl.setFrequency(log.getFrequencyLoop(l));
  loopNametoLoopBS.put(l.getName(), bl);
 }
 //Method to substitute the loop reference with the corresponding block
      structure
 return replaceReferences(loopNametoLoopBS, b);
}
```

Listing B.8: Replacing loop reference method

At last, we generated a block structure for a single event log that can be retrieved using the **getFinalModel()** method of the SchimmAlgorithm class.

B.2 Aggregation

We can use the aggregate function to analyse more than one event log and generate an overall system specification.

This method applies to a list of $mCRL2$ specification objects, meaning we use the algorithm explained above for all the event log separately. Once we have $mCRL2$ object for each event log, we can unify their actions and messages, and the allow, hide and communication sets. Every initial process is added to a common init set that will be represented as the parallel composition of initial processes.

```
public static String mergeMCRL2(List<MCRL2> mcrl2list) {
MCRL2 unicspec = new MCRL2();
mcrl2list.forEach(l -> {
unicspec.addActSet(l.getActSet());
if (l.getAllowedAction().isEmpty())
 unicspec.addAllowedAction(l.getActSet());
else
//Union of act, comm, allow and hide sets
```

```
 unicspec.addAllowedAction(l.getAllowedAction());
 unicspec.addHideAction(l.getHideAction());
 unicspec.addCommFunction(l.getCommFunction());
 unicspec.addInitSet(l.getInitSet());
 unicspec.appendMessage(l.getMessage());
 unicspec.addProcSpec(l.getProcspec());
});
//A communication function among events with the same message is generated
for (Entry<Event, Collection<Event>> m :
      unicspec.getMessage().asMap().entrySet()) {
 Event[] a = new Event[m.getValue().size()];
 unicspec.addCommFunction(m.getValue().toArray(a), m.getKey());
 unicspec.addActSet(m.getKey());
 unicspec.addAllowedAction(m.getKey());
 for(Event e : a)
     unicspec.removedAllowedAction(e);
}
return generateMcrl2File(unicspec);
}
```

Listing B.9: Aggregation

B.3 Running Example

The following listings report each $mCRL2$ specification obtained from the event logs corresponding to each participant of the running example in Figure 24.

```
act
BookTravel,PayTravel,Paymentconfirmationreceived,Bookingconfirmed;
proc
P0=(BookTravel.Bookingconfirmed.PayTravel.Paymentconfirmationreceived);
init P0;
```

Listing B.10: $mCRL2$ specification generated from the Customer log.

```
act
Confirmpayment,TicketOrderReceived,Paymentrefund;
proc
P1=(TicketOrderReceived.(Paymentrefund+Confirmpayment));
init P1;
```

Listing B.11: $mCRL2$ specification generated from the Airline log.

```
act
Orderticket,t,Bookingreceived,Paymentreceived,t0,ConfirmBooking;
proc
P2=((Bookingreceived.ConfirmBooking.t0.Paymentreceived.t0)
     ||(t0.Orderticket.t0));
```

```
init hide({t},
    allow({Orderticket,t,Bookingreceived,Paymentreceived,ConfirmBooking},
            comm({t0|t0->t},
             P2)));
```

Listing B.12: $mCRL2$ specification generated from the Travel agency log.

Next follows the specification of the aggregated logs.

```
act
Confirmpayment,BookTravel,Bookingreceived,Paymentreceived,Paymentrefund,
Bookingconfirmed,ConfirmBooking,confirmation,Orderticket,
TicketOrderReceived,PayTravel,t,Paymentconfirmationreceived,payment,
payment_confirmation,t0,order,travel;
proc
P0=(TicketOrderReceived.(Paymentrefund+Confirmpayment));
P1=(BookTravel.Bookingconfirmed.PayTravel.Paymentconfirmationreceived);
P2=((Bookingreceived.ConfirmBooking.t0.Paymentreceived.t0)
     ||(t0.Orderticket.t0));
init
hide({t},allow({Paymentrefund,confirmation,t,payment,payment_confirmation,
                 order,travel},
  comm({Bookingreceived|BookTravel->travel,
          Confirmpayment|Paymentconfirmationreceived->payment_confirmation,
          Orderticket|TicketOrderReceived->order,
          PayTravel|Paymentreceived->payment,
          Bookingconfirmed|ConfirmBooking->confirmation,t0|t0->t},
     P0||P1||P2)));
```

Listing B.13: $mCRL2$ aggregate specification of the running example.

B.4 Replicate Experiments

The tool is equipped with a way to replicate part of the experiments presented in Section 4.4. The execution of PALM generates a $mCRL2$ specification measuring the execution time and its MC-fitness value, given the same log is able to check which type of equivalence exists between the specification and the coverability graph, then transformed in a $mCRL2$ specification as well, obtained from the other algorithms.

Still, the integration on how to generate the coverability graph needs to be included, given an event log applying the IM, sHM and SM algorithms. For the moment, to replicate this step, the user should follow these steps:

- Download and extract the logs.zip archive from the GitHub repository.

- Execute the TKDE benchmark tool using this line on the terminal.

```
        java -jar tkde\_benchmark\_v2.0.jar -ext ".\logs-folder"
            -miners 0 8 2 -metrics 0
```

  The output will be a BPMN model for each log in the folder and algorithm used, and the csv file contains the execution time.

- For each BPMN generated, use the plug-ins in the ProM framework in the following order:

  1. "Convert BPMN diagram to Petri net"
  2. "Construct coverability graph of a Petri Net"

- The result will be a file ".sg" containing the coverability graph. Download the file inside the folder corresponding to its eventlog inside the "logs" folder and rename it as *namelog_namealgorithm.sg* (example: log1_IM.sg).

- Execute the PALM tool and select "[3] Repeat experiments", and wait for the "result.csv" to be generated.

Bibliography

[1] *4TU*. `https://data.4tu.nl/repository/collection:event_logs_real`.

[2] van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.

[3] van der Aalst et al. "Process Mining Manifesto". In: *Business Process Management Workshops*. Vol. 99. LNBIP. Springer, 2011, pp. 169–194.

[4] Fides Aarts and Frits Vaandrager. "Learning I/O automata". In: *International Conference on Concurrency Theory*. Springer. 2010, pp. 71–85.

[5] Rafael Accorsi, Andreas Lehmann, and Niels Lohmann. "Information leak detection in business process models: Theory, application, and tool support". In: *Information Systems* 47 (2015), pp. 244–257.

[6] Simone Agostinelli et al. "Achieving GDPR Compliance of BPMN Process Models". In: *Information Systems Engineering in Responsible Information Systems - CAiSE Forum 2019, Rome, Italy, June 3-7, 2019, Proceedings*. 2019, pp. 10–22.

[7] Amir Shayan Ahmadian, Daniel Strüber, and Jan Jürjens. "Privacy-enhanced system design modeling based on privacy features". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2019, pp. 1492–1499.

[8] Najla Omrane Aissaoui et al. "A BPMN-VSM based process analysis to improve the efficiency of multidisciplinary outpatient clinics". In: *Production Planning & Control* (2022), pp. 1–31.

[9] Bader K AlNuaimi, Mohammed Al Mazrouei, and Fauzia Jabeen. "Enablers of green business process management in the oil and gas sector". In: *International Journal of Productivity and Performance Management* 69.8 (2020), pp. 1671–1694.

[10] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and computation* 75.2 (1987), pp. 87–106.

[11] Apromore. *See the difference between perception and reality*. 2023. URL: `https://apromore.com/`.

[12] Adriano Augusto et al. "Automated discovery of structured process models from event logs". In: *Data & Knowledge Engineering* 117 (2018), pp. 373–392.

[13] Adriano Augusto et al. "Split miner: Discovering accurate and simple business process models from event logs". In: (2017), pp. 1–10.

[14] Adriano et al. Augusto. "Automated discovery of process models from event logs: Review and benchmark". In: *IEEE transactions on knowledge and data engineering* 31.4 (2018), pp. 686–705.

[15] Ahmed Awad, Gero Decker, and Niels Lohmann. "Diagnosing and Repairing Data Anomalies in Process Models". In: *Business Process Management Workshops*. Springer, 2010, pp. 5–16.

[16] Vanessa Ayala-Rivera and Liliana Pasquale. "The Grace Period Has Ended: An Approach to Operationalize GDPR Requirements". In: *26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018*. 2018, pp. 136–146.

[17] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[18] Sara Belluccini et al. "PALM: A Technique for Process ALgebraic Specification Mining". In: *International Conference on Integrated Formal Methods*. Springer. 2020, pp. 397–418.

[19] Sara Belluccini et al. *PALM: a technique for Process ALgebraic specification Mining (Technical Report)*. Tech. rep. Available at `https://github.com/SaraBellucciniIMT/PALM`. IMT.

[20] Sara Belluccini et al. "Verification of Privacy-Enhanced Collaborations". In: *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*. 2020, pp. 141–152.

[21] Jan A Bergstra and Jan Willem Klop. "Process algebra for synchronous communication". In: *Information and control* 60.1/3 (1984), pp. 109–137.

[22] Ivan et al. Beschastnikh. "Inferring models of concurrent systems from logs of their behavior with CSight". In: *Proceedings of the 36th International Conference on Software Engineering*. IEEE, 2014, pp. 468–479.

[23] Chiara Bodei et al. "Techniques for security checking: non-interference vs control flow analysis". In: *ENTCS* 62 (2002), pp. 211–228.

[24] Egon Börger and Bernhard Thalheim. "A method for verifiable and validatable business process modeling". In: *Advances in Software Engineering*. Vol. 5316. LNCS. Springer, 2008, pp. 59–115.

[25] Achim D Brucker and Sakine Yalman. "Confidentiality Enhanced Life-Cycle Assessment". In: *International Conference on Business Process Management*. Springer. 2021, pp. 434–446.

[26] Joos CAM Buijs, Boudewijn F Van Dongen, and Wil MP van Der Aalst. "On the role of fitness, precision, generalization and simplicity in process discovery". In: *On the Move to Meaningful Internet Systems: OTM 2012: Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part I*. Springer. 2012, pp. 305–322.

[27] Olav Bunte et al. "The mCRL2 toolset for analysing concurrent systems: improvements in expressivity and usability". In: *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part II 25*. Springer. 2019, pp. 21–39.

[28] Andrea Burattin. "PLG2: Multiperspective Process Randomization with Online and Offline Simulations." In: *BPM (Demos)*. 2016, pp. 1–6.

[29] Antonello Calabró, Said Daoudagh, and Eda Marchetti. "Integrating access control and business process for GDPR compliance: A preliminary study." In: *ITASEC*. 2019.

[30] Sara Capecchi et al. "Session Types for Access and Information Flow Control". In: *CONCUR 2010 - Concurrency Theory, 21th International Conference, Paris, France, August 31-September 3, 2010. Proceedings*. Springer, 2010, pp. 237–252.

[31] Josep Carmona et al. "Conformance checking". In: *Switzerland: Springer.[Google Scholar]* 56 (2018).

[32] Celonis. *Process Mining*. 2023. URL: `https://www.celonis.com/process-mining/`.

[33] Ying Chen et al. "Dynamic task offloading for mobile edge computing with hybrid energy supply". In: *Tsinghua Science and Technology* 28.3 (2022), pp. 421–432.

[34] Flavio Corradini et al. "A Guidelines framework for understandable BPMN models". In: *Data Knowl. Eng.* 113 (2018), pp. 129–154.

[35] Flavio Corradini et al. "A Technique for Collaboration Discovery". In: *Enterprise, Business-Process and Information Systems Modeling: 23rd International Conference, BPMDS 2022 and 27th International Conference, EMMSAD 2022, Held at CAiSE 2022, Leuven, Belgium, June 6–7, 2022, Proceedings*. Springer. 2022, pp. 63–78.

[36] Flavio Corradini et al. "BProVe: a formal verification framework for business process models". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2017, pp. 217–228.

[37] Flavio Corradini et al. "Formalising and animating multiple instances in BPMN collaborations". In: *Information Systems* 103 (2022), p. 101459.

[38] Flavio Corradini et al. "RePROSitory: a Repository platform for sharing business PROcess models and logS." In: *ITBPM@ BPM*. 2021, pp. 13–18.

[39] Sjoerd Cranen et al. "An overview of the mCRL2 toolset and its recent advances". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. LNTCS, 7795. Springer. 2013, pp. 199–213.

[40] Fabio D'Agostino et al. "Development of a multiphysics real-time simulator for model-based design of a DC shipboard microgrid". In: *Energies* 13.14 (2020), p. 3580.

[41] George Danezis et al. “Privacy and data protection by design-from policy to engineering”. In: *arXiv preprint arXiv:1501.03726* (2015).

[42] Rocco De Nicola. “A gentle introduction to process algebras”. In: *Notes* 7 (2014).

[43] Rocco De Nicola. “Behavioral equivalences”. In: (2011). Ed. by David Padua, pp. 120–127.

[44] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. “Semantics and analysis of business process models in BPMN”. In: *Information and Software Technology* 50.12 (2008), pp. 1281–1294.

[45] Boudewijn et al. Dongen. “The ProM framework: A new era in process mining tool support”. In: *PETRI NETS*. Springer. 2005, pp. 444–454.

[46] Marlon Dumas et al. *Fundamentals of business process management*. Vol. 1. Springer, 2013.

[47] Marlon Dumas et al. “Multi-level privacy analysis of business processes: the Pleak toolset”. In: *International Journal on Software Tools for Technology Transfer* (2021), pp. 1–21.

[48] Nissreen A. S. El-Saber and Artur Boronat. “BPMN Formalization and Verification using Maude”. In: *Proceedings of the 2014 Workshop on Behaviour Modelling - Foundations and Applications, BM-FA 2014, York, United Kingdom, July 22-22, 2014*. 2014, pp. 1–12.

[49] Intidhar Essefi, Hanene Boussi Rahmouni, and Mohamed Fethi Ladeb. “Integrated privacy decision in BPMN clinical care pathways models using DMN”. In: *Procedia Computer Science* 196 (2022), pp. 509–516.

[50] Michael Fellman and Andrea Zasada. “State of the Art of Business Process Compliance Approaches: A Survey”. In: *Information Systems*. 2014.

[51] Qiang et al. Fu. “Contextual analysis of program logs for understanding system behaviors”. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 397–400.

[52] Hubert Garavel, Maurice H ter Beek, and Jaco van de Pol. “The 2020 expert survey on formal methods”. In: *Formal Methods for Industrial Critical Systems: 25th International Conference, FMICS 2020, Vienna, Austria, September 2–3, 2020, Proceedings 25*. Springer. 2020, pp. 3–69.

[53] Michael Glöckner et al. "Privacy preserving BPMS for collaborative BPaaS". In: *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE. 2017, pp. 925–934.

[54] Heerko Groefsema and Doina Bucur. "A survey of formal business process verification: From soundness to variability". In: *Business Modeling and Software Design*. 2013, pp. 198–203.

[55] Jan Friso Groote and M Mousavi. *Modeling and analysis of communicating systems*. MIT press, 2014.

[56] Jan Friso Groote and M Mousavi. *Modelling and analysis of communicating systems*. Technische Universiteit Eindhoven, 2013.

[57] Jan Friso Groote et al. "The formal specification language mCRL2". In: *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2007.

[58] Christian W Günther and Eric Verbeek. "XES standard definition". In: *Fluxicon Lab.* (2014).

[59] Christian A Hammerschmidt, Radu State, and Sicco Verwer. "Human in the Loop: Interactive Passive Automata Learning via Evidence-Driven State-Merging Algorithms". In: *arXiv preprint arXiv:1707.09430* (2017).

[60] Christian Albert Hammerschmidt et al. "Interpreting Finite Automata for Sequential Data". In: *arXiv preprint arXiv:1611.07100* (2016).

[61] Charles Antony Richard Hoare. "Communicating sequential processes". In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[62] Gerard J Holzmann. "Design and validation of protocols: a tutorial". In: *Computer networks and ISDN systems* 25.9 (1993), pp. 981–1017.

[63] Sara Houhou et al. "A first-order logic semantics for communication-parametric BPMN collaborations". In: *International Conference on Business Process Management*. Springer. 2019, pp. 52–68.

[64] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT algorithm: a redundancy-free approach to active automata learning". In: *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*. Springer. 2014, pp. 307–322.

[65] David N Jansen et al. "An O (m log n) algorithm for branching bisimilarity on labelled transition systems". In: *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II*. Vol. 12079. LNTCS. Springer. 2020, pp. 3–20.

[66] Dionisis Kandris et al. "Applications of wireless sensor networks: an up-to-date survey". In: *Applied System Innovation* 3.1 (2020), p. 14.

[67] Gerhard Keller, August-Wilhelm Scheer, and Markus Nüttgens. *Semantische Prozeßmodellierung auf der Grundlage" Ereignisgesteuerter Prozeßketten (EPK)"*. Inst. für Wirtschaftsinformatik, 1992.

[68] Abdul Ghaffar Khan et al. "A journey of WEB and Blockchain towards the Industry 4.0: An Overview". In: *2019 International Conference on Innovative Computing (ICIC)*. IEEE. 2019, pp. 1–7.

[69] Bartek Kiepuszewski, Arthur Harry Maria ter Hofstede, and Christoph J. Bussler. "On structured workflow modelling". In: Springer, 2000, pp. 431–445.

[70] Leonard Kleinrock. "Distributed systems". In: *Communications of the ACM* 28.11 (1985), pp. 1200–1213.

[71] Ryszard Koniewski, Andrzej Dzielinski, and Krzysztof Amborski. "Use of Petri Nets and Business Processes Management Notation in Modelling and Simulation of Multimodal Logistics Chains". In: *20th European Conference on Modeling and Simulation*. Warsaw: IEEE, 2006, pp. 28–31.

[72] Adarsh Kumar et al. "A novel smart healthcare design, simulation, and implementation using healthcare 4.0 processes". In: *IEEE Access* 8 (2020), pp. 118433–118471.

[73] Sandeep Kumar et al. "Mining message sequence graphs". In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 91–100.

[74] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196.

[75] Ruggero Lanotte et al. "A formal approach to physics-based attacks in cyber-physical systems". In: *ACM Transactions on Privacy and Security (TOPS)* 23.1 (2020), pp. 1–41.

[76] Kristian Bisgaard Lassen and Wil MP van der Aalst. "Complexity metrics for workflow nets". In: *Information and Software Technology* 51.3 (2009), pp. 610–626.

[77] Sander et al. Leemans. "Discovering block-structured process models from event logs:a constructive approach". In: *PETRI NETS*. Springer. 2013, pp. 311–329.

[78] Sander JJ Leemans, Dirk Fahland, and Wil MP van der Aalst. "Discovering block-structured process models from event logs containing infrequent behaviour". In: *International conference on business process management*. Vol. 7927. LNTCS. Springer. 2013, pp. 66–78.

[79] Ann Lindsay, Denise Downs, and Ken Lunn. "Business processes–attempts to find a definition". In: *Inf. Softw. Technol.* 45.15 (2003), pp. 1015–1019.

[80] Slawomir Mandra et al. "Iterative learning control for a class of multivariable distributed systems with experimental validation". In: *IEEE Transactions on Control Systems Technology* 29.3 (2020), pp. 949–960.

[81] Samer Mansour et al. "Wireless sensor network-based air quality monitoring system". In: *2014 international conference on computing, networking and communications (ICNC)*. IEEE. 2014, pp. 545–550.

[82] Cristian Masalagiu et al. "A rigorous methodology for specification and verification of business processes". In: *Formal Aspects of Computing* 21.5 (2009), pp. 495–510.

[83] *mCRL2 - analysing system behaviour*. URL: https://www.mcrl2.org/.

[84] *mcrl22lps*. URL: https://www.mcrl2.org/web/user_manual/tools/release/mcrl22lps.html.

[85] Jan Mendling, Gustaf Neumann, and Markus Nüttgens. "Yet another event-driven process chain". In: *Business Process Management: 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005. Proceedings 3*. Springer. 2005, pp. 428–433.

[86] Jan Mendling, Hajo A. Reijers, and van der Aalst. "Seven process modeling guidelines". In: *Information and Software Technology* 52.2 (2010), pp. 127–136.

[87] R. Milner. *A calculus of communicating systems*. Springer, 1980.

[88] R. Milner. *Communication and Concurrency*. Prentice-Hal, 1989.

[89] Shoichi Morimoto. "A Survey of Formal Verification for Business Process Modeling". In: *Computational Science*. Vol. 5102. LNCS. Springer, 2008, pp. 514–522.

[90] Behrouz Alizadeh Mousavi et al. "A survey of model-based system engineering methods to analyse complex supply chains: A case study in semiconductor supply chain". In: *IFAC-PapersOnLine* 52.13 (2019), pp. 1254–1259.

[91] Kevin P Murphy et al. "Passively learning finite automata". In: Citeseer. 1995.

[92] Rocco De Nicola. "Process Algebras". In: *Encyclopedia of Parallel Computing*. Ed. by David A. Padua. Springer, 2011, pp. 1624–1636.

[93] Marcin Nizioł et al. "Characteristic and comparison of UML, BPMN and EPC based on process models of a training company". In: *Annals of Computer Science and Information Systems* 26 (2021), pp. 193–200.

[94] Olumide Emmanuel Oluyisola et al. "Designing and developing smart production planning and control systems in the industry 4.0 era: a methodology and case study". In: *Journal of Intelligent Manufacturing* 33.1 (2022), pp. 311–332.

[95] OMG. *Business Process Model and Notation (BPMN 2.0)*. 2011. URL: `https://www.omg.org/spec/BPMN/2.0/`.

[96] OMG. *WHAT IS SYSML?* 2023. URL: `https://www.omgsysml.org/what-is-sysml.htm`.

[97] *PALM github repository*. `https://github.com/SaraBellucciniIMT/PALM`.

[98] *PLEAK - Privacy Leakage Analysis Tools*. URL: `https://pleak.io/home`.

[99] Andrea Polini, Andrea Polzonetti, and Barbara Re. "Formal Methods to Improve Public Administration Business Processes". In: *RAIRO - Theor. Inf. and Applic.* 46.2 (2012), pp. 203–229.

[100] Artem Polyvyanyy, Luciano Garcia-Banuelos, and Marlon Dumas. “Structuring acyclic process models”. In: *Information Systems* 37.6 (2012), pp. 518–538.

[101] Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. “Simplified computation and generalization of the refined process structure tree”. In: *International Workshop on Web Services and Formal Methods*. Vol. 6551. LNPSE. Springer. 2010, pp. 25–41.

[102] Davide Prandi, Paola Quaglia, and Nicola Zannone. “Formal Analysis of BPMN Via a Translation into COWS”. In: *Coordination Models and Languages*. Springer, 2008, pp. 249–263.

[103] Davide Prandi, Paola Quaglia, and Nicola Zannone. “Formal Analysis of BPMN Via a Translation into COWS”. In: *COORDINATION*. Vol. 5052. LNCS. 2008, pp. 249–263.

[104] Pille Pullonen, Raimundas Matulevičius, and Dan Bogdanov. “PE-BPMN: privacy-enhanced business process model and notation”. In: *BPM*. Vol. 10445. LNCS. Springer, 2017, pp. 40–56.

[105] Pille Pullonen et al. “Privacy-enhanced BPMN: enabling data privacy analysis in business processes models”. In: *Software and Systems Modeling* 18.6 (2019), pp. 3235–3264.

[106] Mohamed Ramadan, Hicham G. Elmongui, and Riham Hassan. “BPMN formalisation using coloured petri nets”. In: *International Conference on Software Engineering & Applications*. 2011.

[107] Michel Reniers et al. “Completeness of timed μCRL”. In: *Fundamenta Informaticae* 50.3-4 (2002), pp. 361–402.

[108] Alfonso Rodriguez, Eduardo Fernandez-Medina, and Mario Piattini. “A BPMN extension for the modeling of security requirements in business processes”. In: *IEICE transactions on information and systems* 90.4 (2007), pp. 745–752.

[109] Anne et al. Rozinat. “Towards an evaluation framework for process mining algorithms”. In: *BPM Center Report BPM-07-06* 123 (2007), p. 20.

[110] Ahmed Sadik and Christian Goerick. “Multi-Robot System Architecture Design in SysML and BPMN”. In: ().

[111] Mattia Salnitri, Fabiano Dalpiaz, and Paolo Giorgini. “Designing secure business processes with SecBPMN”. In: *Software and Systems Modeling* 16.3 (2017), pp. 737–757.

[112] Koh Song Sang and Bo Zhou. "BPMN security extensions for healthcare process". In: *CIT/IUCC/DASC/PICom*. IEEE. 2015, pp. 2340–2345.

[113] Cleiton dos Santos Garcia et al. "Process mining techniques and applications–A systematic mapping study". In: *Expert Systems with Applications* 133 (2019), pp. 260–295.

[114] SAP. *A Comprehensive Guide to Process Mining: How to Make Better Decisions Faster*. 2023. URL: `https://www.signavio.com/downloads/white-papers/comprehensive-guide-to-process-mining/`.

[115] Guido Schimm. "Mining exact models of concurrent workflows". In: *Computers in Industry* 53.3 (2004), pp. 265–281.

[116] Alexander Seeliger et al. "Process compliance checking using taint flow analysis". In: *ICIS*. Association for Information Systems, 2016.

[117] Laixiang Shan, Xiaomin Du, and Zheng Qin. "Efficient approach of translating LTL formulae into Büchi automata". In: *Frontiers of Computer Science* 9 (2015), pp. 511–523.

[118] Donghwan et al. Shin. "Scalable Inference of System-level Models from Component Logs". In: *arXiv preprint arXiv:1908.02329* (2019).

[119] Alireza Souri, Nima Jafari Navimipour, and Amir Masoud Rahmani. "Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review". In: *Computer Standards & Interfaces* 58 (2018), pp. 1–22.

[120] Ali Sunyaev. "Distributed ledger technology". In: *Internet Computing*. Springer, 2020, pp. 265–299.

[121] Sudeep Tanwar, Karan Parekh, and Richard Evans. "Blockchain-based electronic healthcare record system for healthcare 4.0 applications". In: *Journal of Information Security and Applications* 50 (2020), p. 102407.

[122] Niek Tax et al. "The imprecisions of precision measures in process mining". In: *Information Processing Letters* 135 (2018), pp. 1–8.

[123] *The jBPT library*. URL: `https://github.com/jbpt/codebase`.

[124] Raj Gaurang Tiwari et al. "Exploiting UML diagrams for test case generation: a review". In: *2021 2nd international conference on intelligent engineering and management (ICIEM)*. IEEE. 2021, pp. 457–460.

[125] Aivo Toots et al. "Business Process Privacy Analysis in Pleak". In: *FASE*. Vol. 11424. LNCS. Springer, 2019, pp. 306–312.

[126] Wayes Tushar et al. "Peer-to-peer energy systems for connected communities: A review of recent advances and emerging challenges". In: *Applied Energy* 282 (2021), p. 116131.

[127] UiPath. *Evolve your processes,discover better outcomes*. 2023. URL: `https://www.uipath.com/product/process-mining`.

[128] *UPPAAL*. URL: `https://uppaal.org/`.

[129] Frits W. Vaandrager. "Model learning". In: *Commun. ACM* 60.2 (2017), pp. 86–95.

[130] Wil Van Der Aalst. *Process mining: data science in action*. Vol. 2. Springer, 2016.

[131] Wil Van der Aalst, Ton Weijters, and Laura Maruster. "Workflow mining: Discovering process models from event logs". In: *TKDE* 9 (2004), pp. 1128–1142.

[132] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. "The refined process structure tree". In: *Data & Knowledge Engineering* 68.9 (2009), pp. 793–818.

[133] *Verifying Multi-threaded Software with Spin*. URL: `https://spinroot.com/`.

[134] Sicco Verwer and Christian A Hammerschmidt. "Flexfringe: a passive automaton learning package". In: *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2017, pp. 638–642.

[135] Wattana Viriyasitavat et al. "Blockchain-based business process management (BPM) framework for service composition in industry 4.0". In: *Journal of Intelligent Manufacturing* 31.7 (2020), pp. 1737–1748.

[136] Weijters and Ribeiro. "Flexible heuristics miner". In: *CIDM*. IEEE. 2011, pp. 310–317.

[137] John A. Wise, V. David Hopkin, and Paul Stager. *Verification and validation of complex systems: Human factors issues*. Vol. 110. Springer Science & Business Media, 2013.

[138] Peter Y. H. Wong and Jeremy Gibbons. "Formalisations and applications of BPMN". In: *Sci. Comput. Program.* 76.8 (2011), pp. 633–650.