Hauke Pribnow

Leveraging Propositional Logic-Based Model Checking to Enable Convenient Analysis of Process Models in Arbitrary Graph-Based Process Modeling Languages

Master Thesis

at the Chair for Information Systems and Information Management
(Westfälische Wilhelms-Universität, Münster)

Supervisor: Prof. Dr. Patrick Delfmann

Presented by: Hauke Pribnow
Averkampstraße 9
48151 Münster
+49 151 21149084
hauke.pribnow@uni-muenster.de

Date of Submission: 2017-12-19

Content

Figures V
Tables VI
Listings VII
Abbreviations IX
1 Introduction 1
2 Basic Concepts 3
2.1 Concepts around Analysis of Meta Model-Backed Business Process Models 3
2.2 Concepts around Model Checking 5
3 Model Checking for Meta Model-Backed Business Process Models 10
3.1 Towards Specifying Execution Semantics for Process Modeling Languages 10
3.1.1 Discussion of Existing Approaches 10
3.1.2 Idea of Assigning Behaviors to Model Elements 11
3.1.3 Moving Behaviors to the Meta-Level of Process Models 12
3.2 Making Model Checking Results More Understandable 14
3.2.1 Responsibility-Explaining Model Element Sequences 14
3.2.2 Ability to Map Analysis Results Back to Conceptional Process Models 16
3.3 High-Level Process Logic Transformation and Evaluation Process 17
4 Introducing “Meta Semantics” Languages for Specifying Semantics of Process Models on the Meta Model Level 19
4.1 [em]’s Data Model for Representing Meta Models and Models 19
4.2 Languages’ Data Types and Default Values 21
4.3 Formulaic Expression Language 24
4.3.1 Informal Introduction 24
4.3.2 Syntax 27
4.4 Execution Semantics Description Language 29
5 Foundations for Implementing the Theoretical Approach 35
5.1 Searching and Selecting a Suitable Model Checker 35
5.1.1 Model Checker Requirements 35
5.1.2 Surveying Model Checkers 37
5.2 Generating Formal Processes from [em] Data and Behaviors 38
5.2.1 Translation of [em] Data to LNT 40
5.2.2 Translation of Formulas to LNT 43
5.2.3 Translation of Behaviors to LNT 45
5.3 Making the Model Checker’s Property Specification Language Support Macros 49
5.3.1 Our Macro Extension for the Property Specification Language MCL 50
5.3.2 Translating from Macro-Extended MCL to Plain MCL 51
5.4 User-Perspective Requirements for an Implementation 52
6 Implementing the Approach and Integrating it into [em] 53
6.1 Basic Details on Plugin Implementation 53
6.2 Overview of Plugin Architecture, Persistent Data, and Data Flow 54
6.3 Details on Plugin’s Persistent User Data 55
6.4 Details on Plugin’s Core Components 56
6.4.1 Temporal Property Specification Wizard 56

6.4.2 Formulaic Expression Processor ... 58
6.4.3 Process Model Translator ... 62
6.4.4 Evaluation Preparer ... 63
6.4.5 Evaluation Runner ... 65
6.5 Usability Considerations ... 66
7 Demonstration ... 69
7.1 Case Study 1 ... 69
7.1.1 Introduction into Simple Linear Process Language (SLPL) ... 69
7.1.2 Specification of Behavior Sequences for SLPL ... 70
7.1.3 Model Checking with SLPL Models ... 73
7.2 Case Study 2 ... 75
7.2.1 Basic Highly Simplified EPC without Interfaces (HSPEC) ... 75
7.2.2 Model Checking with HSEPC ... 80
7.3 Case Study 3 ... 83
8 Discussion, Outlook and Summary ... 86
8.1 Comparison with Model Structure-Based Process Model Analysis Approaches ... 86
8.2 Discussion and Outlook on Our Process Model Analysis Approach ... 89
8.2.1 Non-Functional Improvement Potentials ... 89
8.2.2 Functionality-Extending Improvement Potentials ... 91
8.2.3 Conceptual Future Work ... 93
8.3 Summary ... 95
References ... 96
Appendix A Semantics of Formulaic Expression Language ... 102
Appendix B ESDL Formal Specification ... 105
B.a Assignment of Behavior Sequences to Element Occurrences and Models ... 105
B.b Introduction into Our Abstract Machine ... 105
B.c Behavior Types ... 107
B.d Deriving an LTS using Our Abstract Machine ... 110
Appendix C Reference on Data Types in Our Languages ... 112
C.a Boolean ... 112
C.a.a Properties ... 112
C.a.b Functions ... 112
C.b Integer ... 112
C.b.a Properties ... 112
C.b.b Functions ... 113
C.c Double ... 113
C.c.a Properties ... 113
C.c.b Functions ... 114
C.d String ... 114
C.d.a Properties ... 114
C.d.b Functions ... 114
C.e Collection<T> ... 115
C.e.a Properties ... 115
C.e.b Functions ... 115
C.e.c Lambdas ... 115
C.f [em] Data Types ... 116
C.f.a Templates for Main Properties ... 116

C.f.b Additional Element Properties 117
C.f.c Additional ElementOccurrence Properties 117
C.f.d Additional ElementType Properties 117
C.f.e Additional ElementType, ObjectType and RelationshipType Functions 117
C.g Runtime-relevant Types 117
C.g.a Custom Type Properties 118
C.g.b Custom Type Functions 118
Appendix D Source Code of Plugin 119

Figures

Figure 1 An exemplary LTS 7
Figure 2 A reduced version of the LTS as in Figure 1 8
Figure 3 An exemplary process model 11
Figure 4 The process model as in Figure 3 with highlighted elements 15
Figure 5 The high-level workflow of our approach 18
Figure 6 Our perspective on [em]'s data model as an UML Class Diagram 20
Figure 7 Process models for demonstrating unnecessary LTS size increase that occurs when not releasing unused RuntimeInstances 33
Figure 8 Two LTS that can be derived from the process models in Figure 7 34
Figure 9 LNT generation workflow 39
Figure 10 A simple sequence of behaviors as displayed by our implementation 48
Figure 11 The plugin architecture and its high-level data flow 54
Figure 12 Screenshot of our plugin's Temporal Property Specification Wizard 57
Figure 13 Workflow of the Formulaic Expression Processor 58
Figure 14 Screenshot showing our implementation's autocomplete suggestion feature 60
Figure 15 Screenshot showing how a declaration hint is displayed by our plugin 60
Figure 16 The Process Model Translator's workflow 62
Figure 17 The Evaluation Preparer's workflows 63
Figure 18 The Evaluation Runner's workflow 66
Figure 19 Screenshot showing our temporal property specification cheat sheet 67
Figure 20 An exemplary business process for the build preparation of a nuclear reactor 70
Figure 21 Exemplary SLPL model containing a loop 73
Figure 22 Two exemplary counterexample and witness event chains 74
Figure 23 Exemplary highlighting of element occurrences in a witness event chain 75
Figure 24 An exemplary HSEPC model 77
Figure 25 Property Scope page of our Temporal Property Specification Wizard 80
Figure 26 Property Behavior page of our Temporal Property Specification Wizard 81
Figure 27 Event Specification page of our Temporal Property Specification Wizard 81
Figure 28 Counterexample stack for the second case study 82
Figure 29 Highlighted elements relevant for the counterexample of the second case study 82
Figure 30 Section of an exemplary HSEPCwI model with highlighted elements 83
Figure 31 An exemplary SDTL model with highlighted elements 84
Figure 32 Fragment of the witness information of case study 3 84
Figure 33 BPMN model with compensations 87

Tables

Table 1 Exemplary assignments of behaviors to elements of the model in Figure 3 12
Table 2 Exemplary assignment of behaviors to model elements on their meta level 13
Table 3 Informal description of ESDL behavior types 31
Table 4 Members of the default environment for formulaic expressions..... 32
Table 5 Chunks of LNT code to be generated for [em] classes and their instances..... 42
Table 6 Notable aspects on the translation of behaviors to LNT 47
Table 7 Behavior sequence for occurrences of SLPL *Nodes* 71
Table 8 Behavior sequence for occurrences of SLPL *Node Connections*..... 72
Table 9 Behavior sequence for SLPL models 72
Table 10 Behavior sequence for occurrences of HSEPC *XOR* 78
Table 11 Partial behavior sequence for occurrences of HSEPC *AND* 79

Listings

Listing 1 Formula evaluating to an integer value 24
Listing 2 Formula evaluating to a double value 25
Listing 3 Formula evaluating to a string value 25
Listing 4 Formula evaluating to a Boolean value 25
Listing 5 Formula consisting of an identifier 25
Listing 6 Formula with an application of a property accessor on a string 25
Listing 7 Formula with an application of a property accessor on an integer 25
Listing 8 Formula with an application of a single-argument function accessor on an integer 26
Listing 9 Formula with an application of a double-argument function accessor on a string 26
Listing 10 Formula with an application of a lambda accessor 26
Listing 11 Formula with an application of a lambda accessor, using a different parameter name 26
Listing 12 Formula with an accessor chain 26
Listing 13 Formula possibly resulting in *null* result 27
Listing 14 Extension of Listing 13 with a following accessor 27
Listing 15 `Formula` syntax rule 27
Listing 16 `Base` syntax rule 27
Listing 17 Syntax rules for `Base`s 28
Listing 18 `Accessor` syntax rule 28
Listing 19 `PropertyAccessor` syntax rule 28
Listing 20 `FunctionAccessor` and its `ArgumentList` syntax rule 28
Listing 21 `LambdaAccessor` and its `LambdaParameterList` syntax rule 28
Listing 22 Exemplary LNT type declaration for representing [em] Objects 40
Listing 23 LNT type declaration for a nullable list of instances of the [em] Object class 40
Listing 24 LNT function yielding an exemplary list of all [em] Object class instances 41
Listing 25 Exemplary LNT function yielding the value of an Object's Caption attribute 41
Listing 26 Exemplary LNT helper functions yielding instances linked via the "Followers" association for three [em] Objects 42
Listing 27 Exemplary LNT getter function yielding instances linked via the "Followers" association for [em] Objects 42
Listing 28 Formulaic expression to determine the existence of outgoing Relationships from the current ObjectOccurrence 43
Listing 29 LNT expression corresponding to the formulaic expression in Listing 28 44
Listing 30 `custom_count` function for a list of RelationshipOccurrences 44
Listing 31 `>` function for two nullable integers 44

Listing 32 LNT translation of the “Report Event” behavior of Figure 10 48
Listing 33 LNT translation of the “If/Then/Else” behavior of Figure 10 with a placeholder for the LNT translation of its Then Behaviors 49
Listing 34 LNT translation of the “Enable Element Occurrence” behavior of Figure 10 49
Listing 35 Regular expression to capture a macro as introduced by our MCL extension 50
Listing 36 Template of CADP’s string encoding of a LTS transition label entailed by our implementation’s LNT code 51
Listing 37 Substitute template for occurrences of the macro pattern 51
Listing 38 Temporal property in MCL that describes the existence of a cycle in the LTS 74
Listing 39 Formula to check for an incoming relationship of a specific type 76
Listing 40 Event content formula for HSEPC *Functions* with associated *Competent Body* 78
Listing 41 Exemplary property generated by the Temporal Property Specification Wizard 81

Abbreviations

AC	Autocomplete
AST	Abstract Syntax Tree
BCG	Binary Coded Graph
BNF	Backus–Naur form
BPMN	Business Process Modeling Language and Notation
CADP	Construction and Analysis of Distributed Processes
EBNF	Extended Backus–Naur form
EPC	Event-Driven Process Chain
ESDL	Execution Semantics Description Language
HSEPC	Highly Simplified EPC
HSEPCwI	Highly Simplified EPC with Interfaces
LOTOS	Language of Temporal Ordering Specification
LTS	Labeled Transition System
MCL	Model Checking Language
mCRL2	micro Common Representation Language 2
OMG	Object Management Group
RM	Model Checker Requirement
RU	User-Perspective Requirement
SDTL	Simple Decision Tree Language
SLPL	Simple Linear Process Langauge
UC	Usability Consideration
UML	Unified Modeling Language

1 Introduction

Models of business processes capture events that may occur and activities that need to be performed within the context of an organization, often represented in a graphical form. Relying on models of business processes can be risky if these models entail invalid or even harmful sequences of activities.

One such risk is causing injuries to people involved in sectors requiring high security, for example railroad systems (International Union of Railways (UIC) et al. 2009), or nuclear facilities (Lahtinen et al. 2012). Another type of risk is legal prosecution, for example because of violation of regulatory requirements (compliance), e.g. in financial institutions (Becker et al. 2014). Yet another kind of risk is suffering financial losses, e.g. because fraud was not properly prevented or confidentiality was not guaranteed (Arsac et al. 2011), or because of unreasonably handling "edge cases" that were not properly considered during process design. To give an example for such edge cases: An analysis of the SAP reference process models in 2007 found that the models entail activities that possibly lead to financial losses: In the context of procurements, the models elicit payments for goods that were never received. In the context of subcontracting, the process models allow to issue a second payment for some invoice when an identical copy is received a second time. (van Dongen et al. 2007)

When people rely on models of business processes, they may want to reduce or even rule out risks resulting from models entailing invalid or harmful activities. To reduce these risks, users of business process models may want to ensure that the used process models fulfill specific properties that characterize correct and safe processes. We call checking whether a business process model fulfills such properties "business process analysis". One approach for performing business process analysis is a manual one: A person or a group of people manually reviews process models to identify aspects that cause relevant properties to be violated. This may be a feasible solution when a low number of small business processes needs to be analyzed. With higher numbers and larger models, manual analysis may not be feasible anymore. A computer-supported analysis approach may be required in such cases.

Model checking is a concept that may allow supporting business process model analysis with automatic computations. Given a temporal property stated as a temporal logic formula and given a labeled state-transition system, the model checking problem asks to find all states of the system that fulfill the property. Algorithms exist for solving the model checking problem automatically. (Clarke 2008)

Multiple standardized languages exist for representing business process models. For some of such standardized business process modeling languages, model checking approaches were presented, e.g. for BPMN (Raedts et al. 2007) and BPML (Brambilla 2005), or for EPC (van

Dongen et al. 2007). A survey in the financial industry however indicates that a non-negligible amount of processes in practice is modeled with proprietary notations. (Becker et al. 2010)

Meta modeling is an approach that allows to specify conceptual modeling languages (e.g. process modeling languages) and to create models in these languages. (Becker et al. 2004) Since the concept of meta modeling allows to specify custom modeling languages, the concept is not restricted to a specific set of business process modeling languages. Tools exist implementing the concept of meta modeling, for example the tool [em]. (Delfmann et al. 2008)

In this thesis, we present an approach that allows to translate business process model analysis problems into model checking problems and to solve these problems in such a way that the computed results are useful for a business process model analyst. We describe an implementation of our approach as a plugin for [em]. We demonstrate the application of our approach and our implementation in exemplary case studies. We compare our approach with other meta model-based process model analysis approaches, and derive ideas for potential future work from a discussion of the applicability of our approach. The core idea of our approach is defining behaviors for models and elements in a generic fashion on the language level. Generic formulaic expressions describe the conditions which elements should be enabled and which behaviors should be triggered. These conditions are formulated over the attributes of and relationships between models and their elements.

With our approach and our implementation, we provide a solution that easily allows a business process model analyst 1) to specify formal execution semantics for business process models of arbitrary process modeling languages on the language level, 2) to define properties that models with defined execution semantics can be checked for, 3) to automatically solve model checking problems derived from selected properties and models with defined execution semantics, and 4) to derive information from the model checking results that allows understanding which model elements were responsible for the computed result.

The remainder of this thesis is structured as follows. In the second chapter, we introduce basic concepts in detail that form the foundations of our work. In the third chapter, we introduce our general idea and the resulting high-level approach by explaining in natural language how to translate a business process model analysis problem into a model checking problem. In the fourth chapter, we formalize our approach and make it more concrete by introducing additional concepts and languages for describing models' formal execution semantics on a meta-level in an automatically processable form. In the fifth chapter, we lay the foundations for an implementation of our theoretical approach. In the sixth chapter, we describe how we implemented our approach as a plugin for [em]. In the seventh chapter, we demonstrate the applicability of our approach by describing how a set of exemplary artificial case studies can be solved with the implementation of our approach. In the eight chapter, we discuss our results, give an outlook and summarize our work.

2 Basic Concepts

In this chapter, we introduce basic concept that our work is based on. We describe which aspects we put our focus on and in what regard we restrict our perspective. We give definitions and name basic assumptions. Where necessary, we explain why a concept or an assumption is important for our work.

In the first section, we introduce concepts from the domain of meta model-based business process model analysis. In the second section, we introduce concepts from the domain of model checking.

2.1 Concepts around Analysis of Meta Model-Backed Business Process Models

Meta Modeling, Modeling Languages, and Models. "Meta modeling" means using a model to formally define relevant aspects of a modeling language in which models can then be created. A modeling language captures common aspects of the models that are created in this language. (Becker et al. 2004)

Meta modeling provides a way to specify types of elements that a model in the respective language may contain. Among these types may be "object types" and "relationships types". An instance of an object type is called an "object". An instance of a relationship type is called a "relationship". Collectively, objects and relationships are called "elements".

Relationships represent connections between elements in a model. Relationship types specify what and how instances of element types can be put into a relationship with each other. On this basis, we define a model as a subset of the set of elements where each object is an instance of an object type, and relationships are restricted according to the language's relationship type specifications.

The structure of a model as defined here with elements and relationships resembles a graph with nodes and edges, respectively. We therefore call modeling languages that follow our definition "graph-based modeling languages".

The Meta Modeling Tool "[em]. The concept of meta modeling was implemented in software. An example for such an implementation is "[em]". (Delfmann et al. 2008) [em] is a meta modeling tool that allows a user to create a definition of a modeling language and then use this language definition to create models in the defined language. The implementation of a model checking plugin in [em] being one of the core intentions of our work, we focus on [em] in our thesis.

Business Process. The term "business process" has been defined in different ways, for example as a "collection of activities that takes one or more kinds of input and creates an output that is

of value to the customer" (Hammer and Champy 1993), as a "a specific ordering of work activities across time and place, with a beginning, an end, and clearly defined inputs and outputs: a structure for action" (Davenport 1993) or as a "a completely closed, timely and logical sequence of activities which are required to work on a process-oriented business object" (Becker and Schütte 2004) as cited and translated in (Mendling 2007).

For our purposes however, we use a highly simplified definition for a business process: We define it as a sequence of observable events. While most of the aspects that express relevance towards a business need are lost in our definition, we nevertheless use the term "business process" to allow a clear differentiation from other terms containing the word "process" that are introduced and used later, especially "formal process specifications" and LNT `process`es. In subsection 8.2.3, we discuss impacts of this simplification.

Business Process Model. Following (Wikipedia contributors 2017c), we define a business process model as a description of business processes of the same nature that are classified into a model. In other words, a business process model describes or entails possibly occurring sequences of observable events. In this context, an instantiation or an "execution" of a business process model is some business process that is entailed by that model.

Process Modeling Languages. To create business process models in a standardized fashion, various process modeling languages have been proposed, e.g. Business Process Modeling Language and Notation (BPMN) as specified in (Object Management Group 2011), Event-Driven Process Chains (EPC, "Ereignisgesteuerte Prozessketten" in German) as specified in (Keller et al. 1992) or Unified Modeling Language (UML) Activity Diagrams as specified in (Object Management Group 2015), to name just a few. Process modeling languages introduce standardized concepts that process models can be created with, for example the concept of an "activity" or of a "sequence flow".

We call a business process model "meta model-backed" if its process modeling language can be described with a meta model according to our definition. In our work, we focus only on process models in such languages.

Execution Semantics. By itself, a process model is just a graphic or a concept. To give meaning to a process model, "execution semantics" or "operational semantics" is required. Execution semantics describe implicitly or explicitly how a process model must be interpreted, i.e. execution semantics specify what processes are entailed by a process model. As such, execution semantics provides a means to systematically derive possible processes (i.e. possible series of events) from the structure of a process model. (Bolognesi and Brinksma 1987) Formally we define execution semantics as a function with the set of business process models as its domain and the set of sets of sequences of observable events as its codomain.

Some process modeling language specifications explicitly provide execution semantics specifications for models in their respective language – in various degrees of precision and formalism. For all process modeling languages with explicitly given execution semantics known to us, a large part of the execution semantics is defined on the element type level. For example, BPMN's "Activity" element type or EPC's "Function" element type roughly characterizes things that are to be done, and BPMN's "Sequence Flow" element type or EPC's connecting line element type introduce the concept of "flow order" into the process.

For some process modeling languages, it may even possible to define the execution semantics entirely on the element type level of the language. This is not universally true however: BPMN for example allows to define some aspects of models using a natural-language descriptive text, e.g. to annotate under which conditions a Process Flow is to be triggered. This is shown to be a challenge later in this thesis, leading to some restrictions with regard to what process models can meaningfully be checked using our approach.

Process Model Analysis. We define process model analysis as checking if a business process model fulfills given properties. There are several motivational drivers for process model analysis, such as compliance checking, weakness identification, or "semantic soundness" checking. (Becker et al. 2014; Delfmann, Steinhorst, et al. 2015)

Different kinds of properties can be relevant for process model analysis. For example, the number of process model elements or their degree of connectivity might be relevant to estimate the complexity of a process model. The average length of characters in labels might be relevant to estimate its easiness to understand. The positions and sizes of the elements in a graphical model might be relevant to check its well-formattedness.

In our thesis, we focus on temporal properties formulated over the sequences of events entailed by a process model through its execution semantics. We give a more detailed introduction into temporal properties in section 2.2.

2.2 Concepts around Model Checking

Temporal Properties. When referring to process model analysis in our work, we focus on checking the fulfilment of temporal properties. Temporal properties are expressions evaluating to a Boolean value that are formulated over the sequences of events entailed by a business process model.

To make it easier to understand what such a temporal property might be, we give an example: Following (Delfmann and Hübers 2015) we use a provision of the German Geldwäschegesetz (Money Laundering Act) to derive a potentially interesting property from it that one might want a process model to be checked for its fulfillment. Translated from German to English, the first

paragraph of the Geldwäschegesetz reads: “In the context of this law, [the process of] identification consists of (1) the determination of the identity by data collection and (2) the check of the identity.” This provision can be interpreted as: Whenever identification data of a person (e.g. a customer of a bank) is collected, this data must be verified before the data may be used in any other action.

We attempt to re-formulate the exemplary informal legal provision as a temporal property. In natural language the property could be specified as: “In any execution of the process model that is to be checked, the ‘Identification data is used’ event may not occur after the ‘Identification data collected’ event, until an ‘Identification data was successfully verified’ event occurs.” Formulating the property in this way is later shown to be easily formalizable and therefore becomes automatically checkable with our implementation.

From a more general perspective, it is possible to put temporal properties into different classes. Two important basic classes are “liveness properties” and “safety properties”. A liveness property states that an event is required to happen, whereas a safety property states that an event is required to not happen. (Lamport 1977) To give examples for these two classes, we consider the operation of a nuclear reactor. A safety property could be: “Executing the nuclear reactor operation process model should never lead to the occurrence of the event ‘A person near the reactor received a radiation of more than 50,000 μSv’.” A liveness property on the other hand could be: “If the auxiliary feed-water pumps fail, the reactor will shut down within 10 minutes.”

Temporal Property Algebras. To allow for algorithmic processing of temporal properties, a formal way of expressing them is required. Different temporal property algebras were suggested that allow to formally express temporal properties. Notable early temporal algebras were Linear Temporal Logic (LTL) (Pnueli 1977), Computational Tree Logic (CTL) (Clarke and Emerson 1981) and the modal μ-calculus (Kozen 1982; Stirling 1996).

The modal μ-calculus was identified to have a higher expressivity than other proposed algebra and was even described as subsuming “virtually all other temporal logics defined in the literature”. (Mateescu and Thivolle 2008) This high level of expressivity comes with a price: Solving a model checking problem with properties expressed with the modal μ-calculus has a high computational complexity. However, when restricting the modal μ-calculus to an “alternation-free” fragment, model checking problems can be solved in polynomial time without losing expressive power as required in practice. (Emerson and Lei 1986)

Labeled Transition System. A labeled transition system (LTS) is a concept that allows capturing business processes in a formal way. We follow (Mateescu and Thivolle 2008) and define a labeled transition system as a tuple consisting of a set of states S, of a set of labels L, of a set of transitions T where T is a subset of $S \times L \times S$, and of an “initial” state s_0 where s_0 is an element of S.

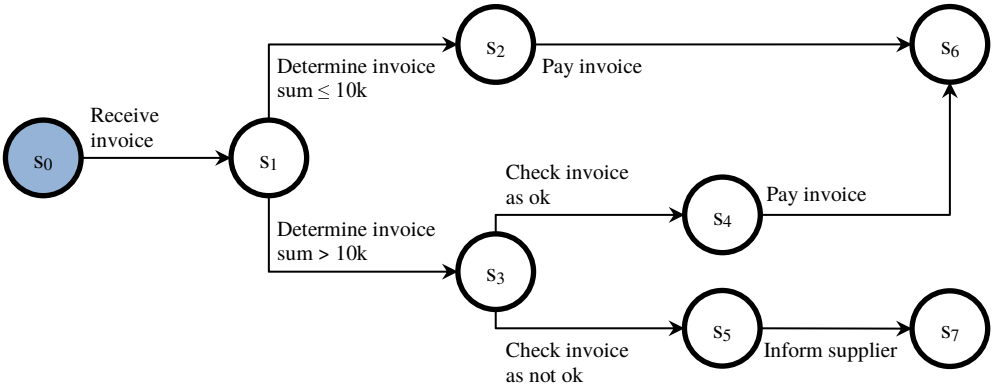

Figure 1 An exemplary LTS

A LTS can be interpreted as a directed graph. In Figure 1, we give a visualization for an LTS that captures the meaning of an exemplary process model that describes how a received invoice needs to be handled. The boxes in the figure represent states and the labeled directed arrows between two boxes represent transitions. The initial state s_0 is highlighted in blue.

To "extract" the underlying business processes from this LTS, we can now start at the initial state and record all series of labels we encounter when recursively following transitions from there. By interpreting each series of labels as a series of events, we have a set of business processes according to our definition.

LTS can be used as a finite description for business processes with sub-sequences of events that are repeated infinitely often. Such processes can be described as an LTS by introducing cycles in the transitions, i.e. sequences of transitions that start at some state and finally return to the same state again.

Reduction of LTS. When working with an LTS in the context of model checking, its number of states and transitions can become an important complexity driver. Therefore, it can be helpful to reduce the number of states and transitions while keeping the captured semantics intact. This can be realized by "reducing" an LTS according to some equivalence relation. Reducing an LTS formally means generating a new LTS so that the new LTS is equivalent to the original LTS according to the given equivalence relation and the new LTS has at most the same number of states and/or of transitions as the original one.

Assuming an equivalence relation that considers two LTS to be equal if they produce the same result when recursively following all possible transitions from the start event and recording the encountered labels, Figure 2 shows the representation of a reduced version of the LTS from Figure 1 according to the equivalence relation. While the original LTS has eight states and eight transitions, the reduced LTS only has five states and seven transitions.

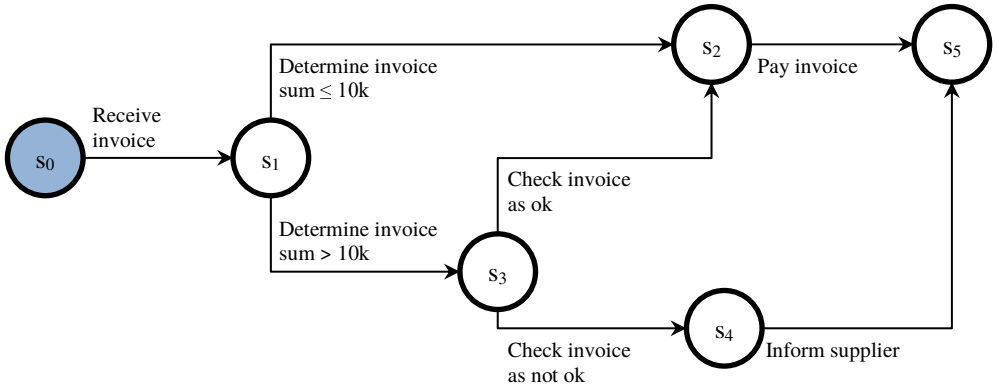

Figure 2 A reduced version of the LTS as in Figure 1

Multiple equivalence relations have been proposed for LTS. The interested reader is suggested to refer to (CADP manual authors 2017d) for more information on some relevant relations.

Formal Process Algebras. When working with business processes that have a lot of parallelism or with business processes requiring an infinite number of states, it may be impractical to work with LTS directly. Formal process algebras were suggested as formal mathematical structures for modeling LTS, i.e. representing an LTS in an abstract form. According to (Technische Universiteit Eindhoven 2017, chap. History), notable formal process algebras were Calculus of Communicating Processes (Milner 1980), Algebra of Communicating Processes (Bergstra and Klop 1984), and Communicating Sequential Processes (Hoare 1978, 1980).

Model Checking and Counterexample / Witness Graphs. Given a model of an LTS and given a temporal property, the "model checking problem" asks to determine if the property is fulfilled by the LTS model. (Clarke 2008) On this basis, model checking is defined as an automatic, cost-effective method for verifying a temporal property of an LTS model, i.e. a method to solve the model checking problem. (Mateescu and Thivolle 2008) An important verification result is a Boolean value that indicates if the temporal property is fulfilled by the model. A "counterexample graph" or a "witness graph" is a second result that can be of interest for some applications.

A counterexample graph is a subgraph of the checked LTS consisting of states and transitions that allow explaining the non-fulfillment of a temporal property. A witness graph is a subgraph of the checked LTS consisting of states and transitions sufficient for explaining the fulfilment of a temporal property. Counterexample graphs can be interpreted as a proof for the model's violation of the property, whereas witness graphs can be interpreted as a proof that something "exists" in the model that was required to exist. Some model checking approaches and implementations allow to compute such counterexample or witness graphs.

Model Checker. A model checker is a tool for solving the model checking problem. In our work, we focus on two model checkers: the *Construction and Analysis of Distributed Processes*

(CADP) model checker and the *micro Common Representation Language 2* (mCRL2) model checker. We describe them in more detail in subsection 5.1.2. The CADP model checker is part our implementation's foundation.

Formal Process Specifications and Formal Process Specification Languages. To make models of LTS automatically processable by a computer, various formal process specification languages based on formal process algebras were proposed. We call a model of an LTS in such a language a "formal process specification".

As one of such languages, the "Language of Temporal Ordering Specification" (LOTOS) was created with the intention to be a formally well-defined standard language that is unambiguous, precise, complete, and implementation-independent. (Bolognesi and Brinksma 1987) LOTOS was further revised in the following years and new languages were created based on different revisions of LOTOS and their underlying concepts. One of these languages is "LNT", originally an abbreviation for "LOTOS New Technology". (Champelovier et al. 2017)

As the CADP model checker accepts models specified in the formal process specification language LNT, we primarily focus on LNT in the remainder of this thesis. In its current implementation, the model checking framework automatically translates LNT to LOTOS and uses the generated LOTOS code for further processing. As such, we also refer to LOTOS to some extent.

Temporal Property Specification Languages. To make temporal properties available for computer-based processing, multiple temporal property specification languages based on temporal property algebras were proposed. In our work, we focus on languages that are based on the modal μ-calculus or a derivative of it because of high expressivity.

One of these temporal property specification languages is the Model Checking Language (MCL) that is supported by the CADP model checker. MCL was proposed as an extension of the alternation-free fragment of μ-calculus with the goal to improving conciseness, readability, and expressiveness of temporal formulas. (CADP manual authors 2017f; Mateescu and Thivolle 2008) We use MCL as the basis for temporal properties that are to be checked with our implementation.

3 Model Checking for Meta Model-Backed Business Process Models

In this chapter, we introduce our general idea in natural language and develop our high-level approach to reach our goal: Given a model, its meta model, and given some temporal property, we want to use model checking to determine if the model fulfills the property and to get a hint why the property is fulfilled or why it is not, respectively. Starting from a model, its meta model and a property, we develop additionally required inputs required to reach the goal.

In the first section, we introduce the foundation of our idea for enabling model checking with meta model-backed business process models. In the second section, we introduce our idea for making the results of model checking more understandable in the context of business process models. In the third section, we bring our ideas together and present our combined overall high-level approach.

3.1 Towards Specifying Execution Semantics for Process Modeling Languages

As introduced in section 2.2, model checking requires two inputs: an LTS and a temporal property. To apply model checking, it is therefore required to derive an LTS from a given model and its meta model. In this section, we review existing approaches for deriving such an LTS and develop our idea behind our approach.

In the first subsection, we discuss the applicability of existing approaches in the context of meta model-backed business process models. In the second subsection, we introduce our idea of assigning behaviors to model elements for describing the model's execution semantics. In the third subsection, we generalize our idea and describe how behaviors can be assigned to elements of a model's meta model.

3.1.1 Discussion of Existing Approaches

One type of approaches to derive a LTS from a business process model is based on pre-defined static language-specific rules for translating a process model into a formal process specification.

For some popular process modeling languages, corresponding formal semantics were specified by the language creators. This applies to BPMN (Object Management Group 2011) and UML Activity Diagrams (Object Management Group 2015) for example. Static pre-defined transformation rules to transform process models into forma process specifications have been proposed and applied at least for BPMN. (Raedts et al. 2007)

For some other popular process modeling languages, only informal semantics were specified by the language authors. This applies to EPC for example. (Keller et al. 1992) For informally

specified semantics it can be difficult to find formal semantics. For EPC it was even shown to be impossible to define sound formal semantics that is fully compliant with the informal semantics. (van der Aalst et al. 2002)

For other process modeling languages, neither formal nor informal semantics may be specified yet. Also, meta modeling allows to create new process modeling languages with new execution semantics that cannot be anticipated in pre-defined rules. The approach of using pre-defined static rules therefore does not work well with the flexible concept of meta modeling.

For this reason, we develop a more flexible approach in our thesis that allows the creation and modification of custom execution semantics.

3.1.2 Idea of Assigning Behaviors to Model Elements

At the core of our approach towards specifying execution semantics we introduce the concept of a "behavior". We further introduce the idea that each model element can be enabled, leading to some behaviors to be triggered. Types of behaviors include "report some event", "enable some model element", "check if some condition is met and – if it is – trigger another behavior", or "choose between multiple behaviors". By assigning behaviors to model elements, the model of LTS is implicitly specified: The events that are reported by behaviors triggered through enablements of model elements become the LTS transitions.

We give an intuition for this concept with an example. Consider Figure 3 for an exemplary process model in EPC where an "XS" element represents an XOR split and an "XJ" represents an XOR join. Each model object has an identifier given in black text next to the respective model element. A sketch of what behaviors could be assigned for this process model is given in Table 1.

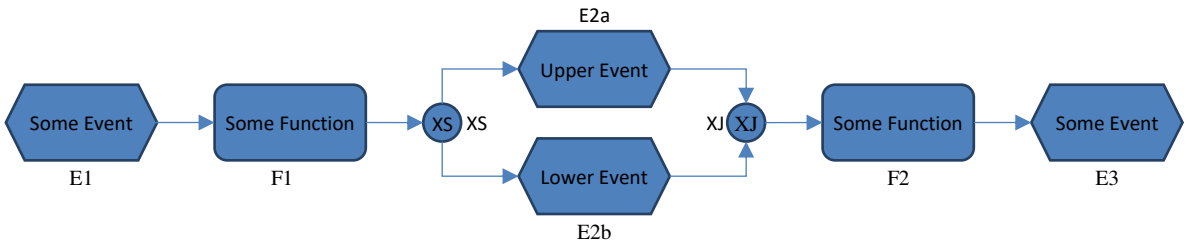

Figure 3 An exemplary process model

Element	Assigned Behaviors
E1	1. Report event “Some Event” 2. Enable element F1
F1	Enable element XS
XS	Choose between the following two behaviors: a. Enable element E2a b. Enable element E2b
E2a	1. Report event “Upper Event” 2. Enable element XJ
E2b	1. Report event “Lower Event” 2. Enable element XJ
XJ	Enable element F2
F2	Enable element E3
E3	Report event “Some Event”

Table 1 Exemplary assignments of behaviors to elements of the model in Figure 3

Using the model elements and the behaviors, we can now follow the sequence of behaviors that are to be triggered when enabling model elements. For now, we assume that we always enable the model element that does not have any “incoming” relationships first, i.e. the one that does not have any predecessors. We lose this assumption later during the design of our actual implementation.

When starting with enabling E1, the event “Some Event” will be reported, followed by enabling the element F1. When F1 is enabled, XS will be enabled next. When XS is enabled, a choice will be made to either enable the element E2a or the element E2b. When enabling E2a or E2b, an event according to the element’s label will be reported, followed by enabling the element XJ. This chain is continued until E3 is enabled and finally the event “Some Event” is reported again, this time without any other element enabling following.

From the possible chains of triggerings, two event sequences can be derived: $(\text{“}Some\ Event\text{”}, \text{“}Upper\ Event\text{”}, \text{“}Some\ Event\text{”})$ and $(\text{“}Some\ Event\text{”}, \text{“}Lower\ Event\text{”}, \text{“}Some\ Event\text{”})$. This demonstrates that introducing behaviors and assigning them to model elements allows to implicitly describe a set of event sequences and therefore fulfills one requirement for model checking.

3.1.3 Moving Behaviors to the Meta-Level of Process Models

Directly assigning concrete behaviors to model elements as proposed in the last section has disadvantages: When analyzing multiple models using model checking, each model would require explicit specification of behaviors for each model element. Simply put, each model would have to be prepared individually, causing a high preparation effort.

Also modifying models would result in additional effort: Assume that a new event model element and a new function model element should be inserted in the process model of Figure 3 “in between” the elements F1 and XS. To get a correct behaviors assignment that fits to the new model, not only new behaviors need to be explicitly specified and assigned to the new elements

but also the existing behavior that is assigned to F1 needs to be updated so that it does not enable XS anymore but the newly introduced event model. Generally speaking, changing a model requires the behavior assignments to be explicitly updated when using the approach of assigning behaviors directly to model elements.

This additional effort would make this naïve approach impractical. A more generic approach would therefore be helpful that involves less effort when changing models or their elements. To create such an approach, we generalize our behavior-based approach in two ways: Instead of specifying a behavior with concrete input data (like E1, F1, E2a, “Some Event”, “Some Function” etc.), we use formulaic expressions to describe the required input data for behaviors in a more generic way (e.g. “successor of current element”, “label of current element”). And instead of assigning behaviors directly to model elements, we associate behaviors with model element types, thereby making use of the meta model to define behaviors in a more generic fashion. With this meta model-based approach, the enablement of a model element of some type now means that those behaviors are to be triggered that are associated with this type.

To give an intuition for this new approach, we pick up the exemplary process model in Figure 3 again and now specify behaviors using generic formulas on meta model level. We give a possible behavior assignment in Table 2. We use curly brackets to indicate generic formulaic expressions (stated in natural language) that play the role of placeholders.

Model Element Type	**Assigned (Generic) Behaviors**
Event	1. Report event {label of current element} 2. If {number of successors of current element greater than 0} then: Enable element {first successor of current element}
Function	Enable element {successor of current element}
XOR Split	1. Choose any one of {successors of current element} and designate it as ‘e’ 2. Enable element {e}
XOR Join	Enable element {successor of current element}

Table 2 Exemplary assignment of behaviors to model elements on their meta level

It is left to the reader to verify that recording reported events from enabling model elements based on the given assignment would result in the same event sequences as given at the end of section 3.1.2 for the model in Figure 3.

Having defined this assignment of behaviors to model element types once, we can now create new or modify existing well-formed EPC models that make use of only the specified four element types without the need for any adjustments to make such models analyzable with model checking.

So far, we have specified the formulaic expressions in natural language. To allow a computer to automatically process such formulaic expressions, a formalized expression language is required that a computer can read and interpret.

As the core idea of our approach, we introduce meta model-based execution semantics as a new input parameter and can now rephrase our overall goal: Given a model, given its meta model, given meta model-based execution semantics specified through behaviors using generic formalized formulas, and given a formal temporal property, we want to use model checking to determine the fulfillment of the given temporal property by the given model, and to get a hint why the property is fulfilled or why it is not, respectively.

3.2 Making Model Checking Results More Understandable

As established in our goal description, we want to use model checking to find out if a property is fulfilled or not and to get a hint why a property is fulfilled or why it is not. Until now we did not specify explicitly what we mean with "getting a hint why a property is fulfilled or why it is not". In this section, we explain what such a hint may be, why it can be useful, and how our approach must be adjusted to be able to produce it.

In the first subsection, we establish the reasons why a hint for a fulfillment or non-fulfilment of a property can be helpful and explain how such a hint may be represented. In the second subsection, we extend the idea developed in section 3.1 to allow deriving such a hint from model checking of meta model-backed business process models.

3.2.1 Responsibility-Explaining Model Element Sequences

The helpfulness of understanding why a property is fulfilled or not fulfilled can be motivated with an example: Take a property that specifies that the event "Upper Event" needs to happen at least once in every process entailed by a given process model. Now assume the exemplary process model from section 3.1 as given in Figure 3 should be analyzed if it satisfies this property.

We have shown earlier that "Upper Event" does not occur for one of the two event sequences entailed by the process model. We therefore already know that the process model does not satisfies the property. Also, in this simple example it may become clear simply from looking at the model why the property is not fulfilled: There is no model element whose enablement would result in a report of the event "Upper Event" on the lower path that branches off from the XOR split in the model. Finding out the reason why a property is not satisfied can however become more difficult for larger and more complex models and properties.

We assume that business process model analysis is usually done to identify needs for changing a model: If an analyzed model does for example not fulfill some property, a business process model analyst may want to change or fix the model so that the new model version does fulfill the property, for example because of legal requirements.

Fixing our exemplary model could be done by inserting a model element in the lower branched-off path that would report "Upper Event" when enabled. So, while it is already helpful to know that the property is not fulfilled, it might be even more helpful to get information that is helpful for fixing the model.

We propose that a sequence of enabled model elements that are "responsible" for the violation (or fulfillment) of the given property may be of help in such situations.

In our example, this can be the sequence $(E1, F1, XS, E2b, XJ, F2, E3)$ or – if restricted to event-reporting model elements only – the sequence $(E1, E2b, E3)$. When following one of these two sequences, the model creator can identify a path through the model where a model element that reports "Upper Event" is missing. In Figure 4, we give a version of the process model as in Figure 3 where the event-reporting model elements that are "responsible" for the violation of the given property are highlighted. Using this visualization, the model analyst can see visually elements that are involved and not involved in the non-fulfillment of the property. This allows easy identification of "problematic" paths.

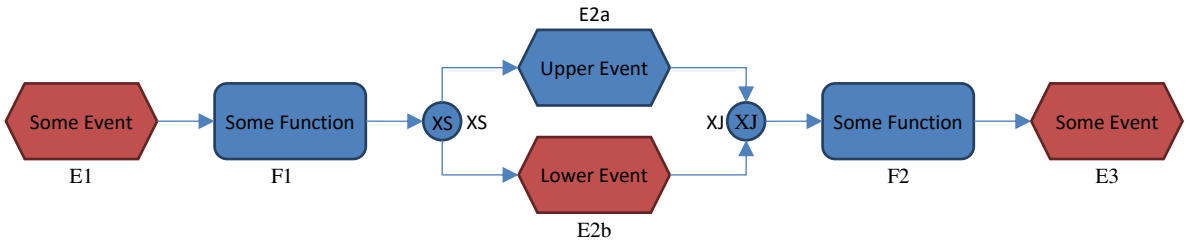

Figure 4 The process model as in Figure 3 with highlighted elements

For some properties, sequences of elements responsible for their violation or fulfillment may not be finite. This applies for example to a property describing the existence of an infinite loop of some event reported by some element. We therefore generalize our proposal to provide sequences of enabled model elements: Instead of providing sequences of elements, we propose to provide subgraphs of elements based on the model's LTS: If a LTS entailed by a process model contains a subgraph that explains the fulfillment or violation of the property, then we assume that a graph of elements corresponding to the LTS subgraph may be helpful for fixing the model according to the requirements that the property is based on.

We assume such graphs of model elements to be of more help for business process analysts than just Boolean values indicating property fulfillment or nonfulfillment. We further assume visualizing the elements in the graph by highlighting them in the model to be even more helpful for such analysts. We call such a graph of model elements and its visualization in the model collectively "counterexample information" or "witness information", respectively.

3.2.2 Ability to Map Analysis Results Back to Conceptional Process Models

In the last subsection, we have established that it can be helpful for a business process analyst to get a graph of model elements that are "responsible" for the model checking result. In this subsection, we explain how our approach from section 3.1 needs to be adjusted to produce such a graph.

As introduced in section 2.2, some model checking approaches and implementations allow to derive counterexample or witness graphs. We pick up our example of section 3.1 again and assume to check our exemplary model given in Figure 3. We check if it satisfies a property that requires "Upper Event" to occur in every execution of the model. If the used model checking approach supports the generation of counterexample and witness graphs, the respective resulting graph would be the path ("Some Event", "Lower Event", "Some Event") as this is the path of events that shows the violation of the property.

If we now want to derive the "responsible" model element sequence from this event sequence, we face a problem with the approach established so far: The events reported through our approach miss any connotation of the model elements that the events were reported for. In our example, a possible solution for the problem of deriving model elements from the events would be identifying a model element from an event by comparing the label of each element with the event. The solution of such an naïve approach could however be ambiguous. We can show this with our example: The event "Some Event" could have been reported both from E1 and from E3. Therefore, this naïve approach is not a generally applicable solution.

We propose an alternative solution. Assume that we have a model element ME that can uniquely identified by identifier i. Further assume that an event E is to be reported for ME. We call E the "public" event. Instead of reporting E, we ensure in our solution that the derived LTS actually reports the "private" event (i, E), i.e. a tuple event consisting of a value that allows identification of both the model element and the original event.

Picking up our example once again using this solution, the respective resulting counterexample graph would now be the path $((E1, \text{“}Some\ Event\text{”}), (E2b, \text{“}Lower\ Event\text{”}), (E3, \text{“}Some\ Event\text{”}))$. With the model element identifiers included in the counterexample graph, it now becomes trivial to derive the graph of "responsible" model elements from it.

By "rewriting" events in the described way, we introduce the requirement of placeholder support for the underlying specification language. Assume that we want to formulate a property that the public event "Some Event" needs to happen twice in every process entailed by a given process model. If this property should now be used in model checking based on a model with rewritten events, the property needs to be rewritten so that it uses private events. A rewritten version of our example property could now read like this: At some point the private event

$(i_1, \text{“}Some\ Event\text{”})$ needs to occur, and then at some later point, the private event $(i_2, \text{“}Some\ Event\text{”})$ needs to occur, for any i_1 and for any i_2 in every process entailed by a given process model, i_1 and i_2 being the two placeholders.

When rewriting events from public to private versions in a specification language, the model element in each private event needs to be a placeholder. Consequently, the used specification language needs to support such placeholders.

While private events can be helpful to get insight when analyzing a model checking result, we assume them to be of little use during property creation. We assume a property creator to consider public events as more interesting than the private ones. To give an intuition for this assumption: It may be more interesting to check if the event “Money was received” will always occur after “Invoice is sent” instead of checking if model element 123 will always be enabled after model element 456.

As such, it may be helpful for property creators if the specification language supports a “macro” that rewrites a given public event to a construct that matches a private event corresponding to the public event and a placeholder for any model element. In our implementation, we provide such an extension for the language supported by the model checker that we use.

3.3 High-Level Process Logic Transformation and Evaluation Process

In section 3.1, we have presented an idea that allows us to specify execution semantics for a model on its meta-level in such a way that we can derive an LTS from a model, its meta model and the specified execution semantics that can be used for model checking. In 3.2 we have presented an event rewriting approach as an extension of our initial idea that allows a user to understand a model checking result better.

In this section, we derive abstract mechanisms from these ideas that process the described inputs to finally generate the wanted outputs. By putting these mechanisms together, we develop our overall abstract approach.

Our overall approach requires the following inputs: 1) A model and its meta model. 2) Formal execution semantics for the meta model. 3) A temporal property formulated in some specification language that might use of event rewriting macros as introduced in subsection 3.2.2.

At the core of operationalizing our approach, we need a model checker that takes in a description of an LTS in a supported specification language and a temporal property in a supported specification language. This model checker needs to generate a Boolean result indicating if the given property is fulfilled by the given model and should generate a counterexample or witness graph.

To generate the LTS description for the model checker, we need a transformer that takes in a model, its meta model and the meta model-based execution semantics. The transformer needs to generate a LTS description in a formal process specification language that is supported by the model checker.

To generate the temporal property, we need a macro expander that takes in a temporal property in a model checker-supported specification language that was extended with a macro construct as introduced in subsection 3.2.2. The macro expander should expand each macro so that the generated property formulation complies with the non-extended, standard version of a specification language that is supported by the model checker.

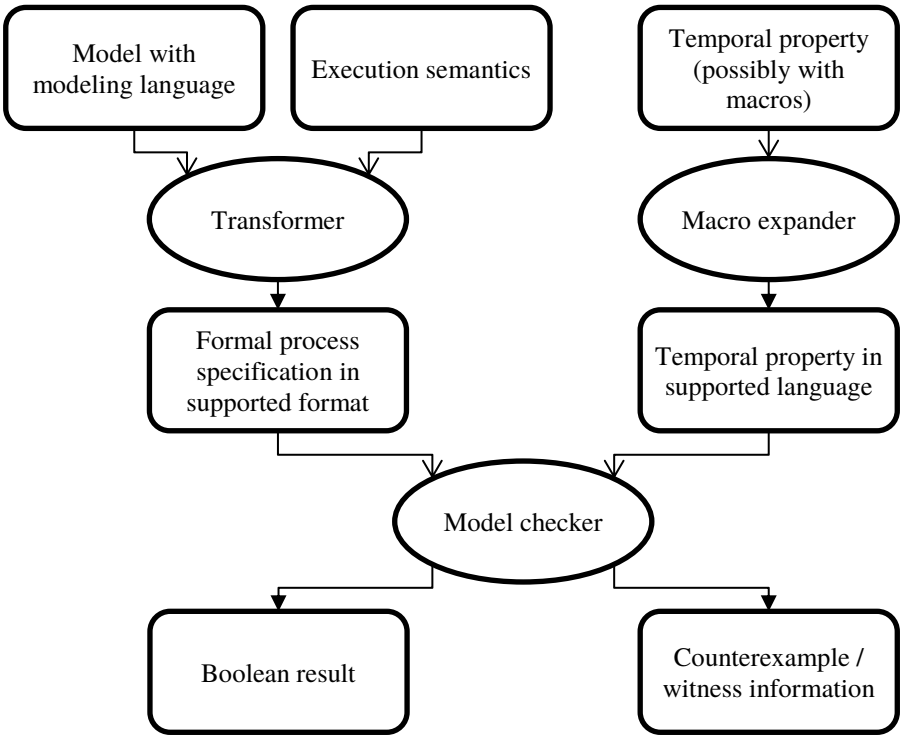

Figure 5 The high-level workflow of our approach

With the employed mechanisms introduced, our overall abstract data processing approach can now be explicitly stated: Provide a model, its meta model and meta model-based execution semantics to the transformer. Provide a temporal property that can contain unexpanded macros to the macro expander. Provide the formal process generated by the transformer and the macro-expanded temporal property to the model checker. The final results of our approach are the two model checker's outputs.

A graphical representation of the overall data processing approach is given in Figure 5. We use boxes with rounded corners to represent data, and ovals to represent components that process data. Connections with open arrows represent that the source element is an input for the target element. Connections with closed arrows represent that the source element provides the target element as output.

4 Introducing “Meta Semantics” Languages for Specifying Semantics of Process Models on the Meta Model Level

In this chapter, we formalize our high-level approach and refine details of it by describing and introducing formal concepts that can be used in an implementation. We introduce “meta semantics” languages, i.e. languages allowing to define execution semantics for a model on its meta-level.

The idea behind our meta semantics languages is based on assigning information to instances of types in a data model that allows to represent meta models (i.e. modeling languages) and their instances (i.e. models). The general idea behind our languages is not dependent on a specific data model and we expect it to be implementable with different data models that allow representing modeling languages and models.

We nevertheless base the descriptions and specifications of our languages specifically on the data model of [em], for two reasons: 1) The descriptions and specifications are more concise when reducing the level of abstraction, especially because the domain of meta modeling is filled with many abstractions already. 2) Our work has the goal of implementing our approach in a plugin for [em], so selecting [em]’s data model as the base of our languages makes them more directly implementable. If our approach should be implemented with a different base data model than [em]’s one, it would have to be adopted accordingly.

In the first section, we present the [em] data model that allows representing meta models and models. In the second section, we introduce the data types supported by our languages. In the third section, we introduce our formulaic expression language that allows to formulate model checking-relevant values in an abstract way. In the fourth section, we introduce our Execution Semantics Description Language that allows to formulate behavior sequences that can be assigned to models and their elements on a meta-level to describe a model’s execution semantics.

Since this section introduces formalisms that may be difficult to understand without examples, it may be helpful to read this chapter in parallel to chapter 7 where we present examples through our case studies.

4.1 [em]’s Data Model for Representing Meta Models and Models

In this section, we present the data model of [em] for representing meta models (i.e. modeling languages) and their instances (i.e. models).

An abstract description of this data model is given in (Delfmann et al. 2008). The concreteimplementation of [em] however diverges from this abstract description to some extent.

As our intended implementation of our approach needs to operate with [em] data as provided by the implementation of [em] and not by its abstract description, we adopt a perspective for our description that is closer to the implementation of the tool.

We present our perspective on [em]'s data model as an UML Class Diagram as given in Figure 6. All composition and aggregation associations in the diagram are to be interpreted as one-to-many relationships, whereas all other associations are to be interpreted as many-to-many relationships. In the remainder of this section, we explain the shown aspects shortly.

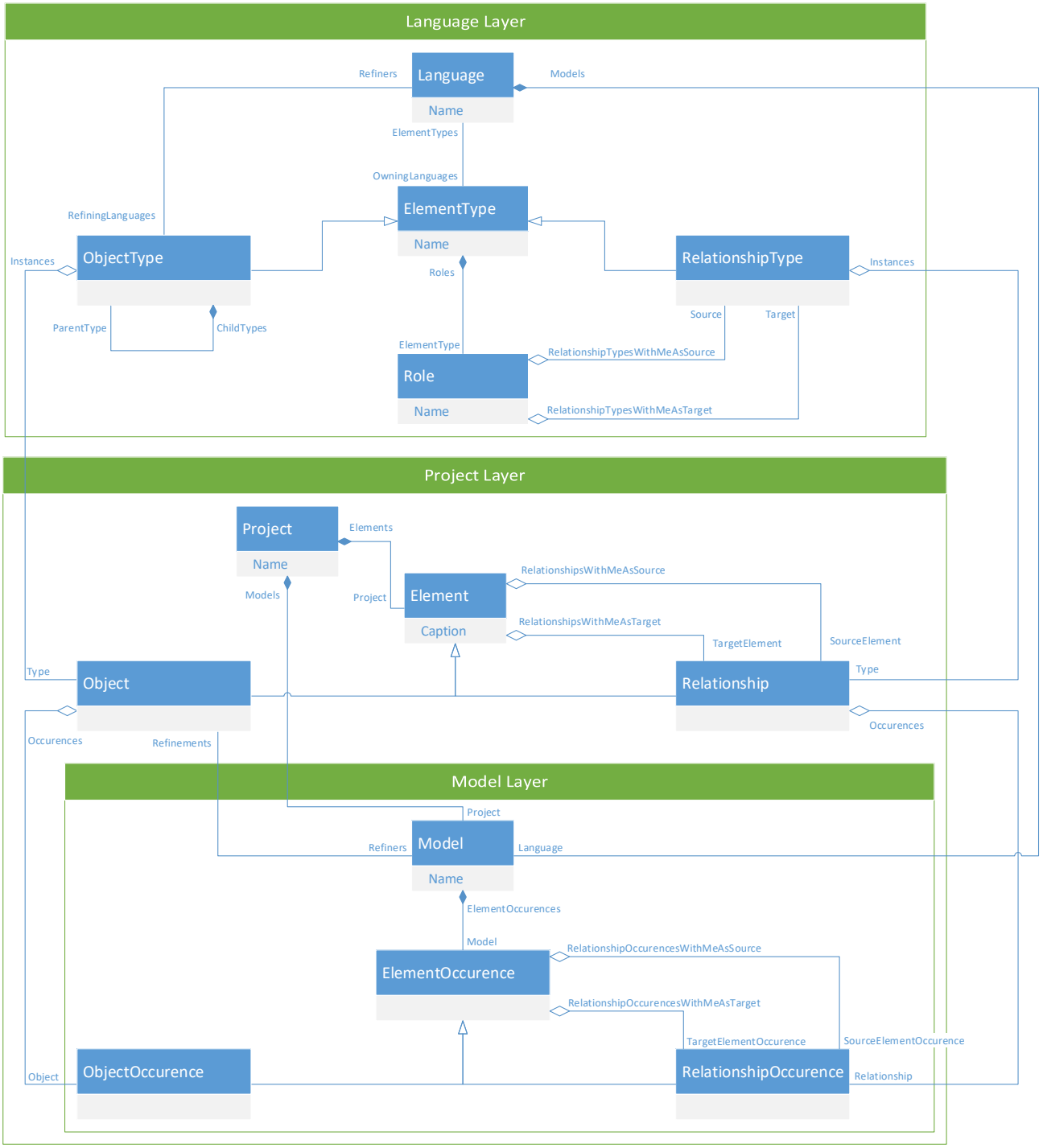

Figure 6 Our perspective on [em]'s data model as an UML Class Diagram

A *Language* in [em] consists of *ElementTypes*. An *ElementType* is either a *ObjectType* or a *RelationshipType*. *ObjectTypes* can be used to form an object inheritance hierarchy using the association “ChildTypes”/“ParentType”.

ObjectTypes and *RelationshipTypes* can be instantiated as *Objects* and *Relationships*, respectively. Collectively, *Objects* and *Relationships* are *Elements*. Each *Element* belongs to some *Project*.

An *Element* can play different *Roles* as defined for its type. On this basis, a *RelationshipType* is specified in such a way that its instances can only be formed between *Elements* whose types agree with the specified *Roles*.

Models can be created in a *Project*. A *Language* is specified for each *Model*. A *Model* contains *ElementOccurrences* of the instantiated *Elements* in the *Model*’s *Project*. An *ElementOccurrence* is either an *ObjectOccurrence* or a *RelationshipOccurrence* that is an occurrence of some *Object* or *Element*, respectively. It is possible to refer to the same *Element* multiple times in the same *Model* by having multiple occurrences of the respective *Element*.

An *ObjectType* can allow its instances to be “refined” by *Models* in specific *Languages*. An instance of an *ObjectType* allowing refinements can then be associated with *Models* of the allowed *Languages*. The idea behind refinements can be explained with an example: Consider two process models where the first is an abstract, generic description how to work in word processing applications, and the second one is a detailed description what mouse clicks and key strokes must be performed to execute the “save” command in the application. Assume one object in the first process model refers to saving the document. Assume that this object is labelled with “Save document”. Then the second process model can be said to “refine” the “Save document” object of the first model because the whole second model provides a more detailed view on the respective single object of the first model.

The full data model of [em] allows *Objects* to not just carry a name but also additional values of specific types. For simplicity, we do not take values carried by *Objects* into account in our work; such value-carrying *Objects* are not required for demonstrating that our idea can successfully be implemented and used. In section 8.2.2, we explain that it would be possible to extend our approach and our implementation to also support values carried by *Objects*.

4.2 Languages’ Data Types and Default Values

In this section, we introduce and formally define the data types that are supported by our languages. First, we define several data types that we call “primitive types”. Second, we specify the set of [em] data types. Third, we introduce data types that ElementTypes of an [em]

Language can be referenced with. Fourth, we introduce a set of data types that we call "runtime-relevant types". Finally, we introduce the concept of default values for several of the data types.

Primitive Types. Primitive types of our language are Boolean, integer, double, string, and collection.

Boolean values in our language are elements of the set $Bool = \{ true, false, null \}$.

Integer values are elements of the set $Integer = \{ -(2^b), -(2^b) + 1, \dots, (2^b) - 1, null \}$ where b is the bit length that the underlying system is set to use. In our current implementation, b is set to 16.

Double values in our implementation are floating-point numbers as realized by the `double` implementation of the underlying C compiler, or $null$. The C11 standard promotes the double format as presented in (International Organization for Standardization 1989) to realize `double`. (International Organization for Standardization 2011) Assuming modern C compilers compiling for modern hardware follow this proposal, double values in our implementation are typically 64-bit floating-point values or $null$. We denote the set of all possible double values with $Double$.

String values are elements of the set $String = \bigcup_{n \in N_0} C^n \cup \{null\}$ where C is the set of all characters supported by the underlying architecture. The underlying model checker of our implementation supports the ASCII character set as specified in (American National Standard for Information Standards 1986). (Champelovier et al. 2017, pp. 25–26)

Collection values for a type T are elements of the set that encompasses $null$ and all sequences of elements in T. We denote the set of all possible collection values for a type T with $Collection\langle T \rangle$. We use the term "members" to refer to the elements in a collection's element sequence.

[em] Data Types. Each class in the data model of [em] as introduced in section 4.1 becomes an [em] data type of our language. Instances of these classes become elements of the respective type sets in our language together with the element $null$. We denote the set of instances of an [em] class and $null$ with its [em] class name in italics. For example, the set $ElementOccurrence$ encompasses $null$ and all [em] Element Occurrences that are available in [em] when model checking processes start.

Runtime-relevant types. Runtime-relevant types of our language are RuntimeInstance, CustomEnablementData, and CustomStorageData.

RuntimeInstance values are elements of the set $RuntimeInstance = \{ri_0, ri_1, \dots\}$. Intuitively, a RuntimeInstance is a storage frame for a set of behaviors that need to keep their stored

information separate from other behaviors. This is especially helpful for languages that use model-overarching behavior chains: If a behavior of a model element in one model enables an element in another model, then it might be required for the two models to keep the data stored for their elements separated from one another. In such a case, both models could maintain their individual RuntimeInstances. The concept of RuntimeInstances is described in more detail in section 4.4.

The other two additional runtime-relevant types, namely CustomEnablementData and CustomStorageData, are to be specified by a user before model checking processes start.

CustomEnablementData is a type for data that is passed to an [em] element's or model's behavior when it is enabled as described in section 4.4. Passing data to an [em] element for example allows to keep track over some source element in case this element must be enabled again later. This is especially helpful for implementing jumps from one process model to another and back.

Intuitively, CustomEnablementData is a struct, i.e. a collection of fields. A language user specifies its fields, each with a name and a type. Our current implementation allows to specify fields of the types Boolean, integer, double, string, RuntimeInstance, Model, and Element Occurrence.

Formally, $CustomEnablementData$ is the product of types as selected by a language user from the list $Boolean$, $Integer$, $Double$, $String$, $RuntimeInstance$, $Model$, $ElementOccurence$. The fields names can be captured as a sequence of identifiers that has equal length equal as the number of terms in the type product.

CustomStorageData is a type of data that is to be temporarily stored for a RuntimeInstance and an ElementOccurrence. Storing data for example allows to keep track over the number of times an ElementOccurrence has already been enabled. This is especially helpful for implementing the Simple Merge workflow pattern as specified in (van der Aalst and ter Hofstede 2017; van der Aalst et al. 2003).

CustomStorageData can be interpreted as an extension of CustomEnablementData. Its definition is equal to CustomEnablementData with the exception that it also allows to specify CustomEnablementData as field type.

Default Values. For data types that can be used for fields in CustomEnablementData types and CustomStorageData types, we define a default value. When one of these two types is instantiated, its fields will have their types' respective default value. We define the default values in the function $Default$ that maps a data type to its default value:

$$Default(Boolean) = true,$$

$Default(Integer) = 0$,

$Default(Double) = 0.0$,

$Default(String) = ""$,

$Default(RuntimeInstance) = ri_0$,

$Default(CustomStorageData) = (f_0, \dots, f_n)$ where $f_i = Default(ft_i)$ and ft_i is the type of the ith CustomStorageData field.

4.3 Formulaic Expression Language

In subsection 3.1.3 we have established that a formulaic expression language can be helpful to specify inputs for behaviors. In this section, we introduce a concrete expression language that can be used for this purpose.

In the first subsection, we give an informal introduction into the language. In the second subsection, we specify the language's syntax. Due to space constraints, we give a formal specification for the language's semantics only in Appendix A.

4.3.1 Informal Introduction

In this subsection, we informally introduce our formulaic expression language. We first give general information explaining the design of our language. We then give example formulaic expressions and explain their meaning to convey a "feeling" for the language.

Early during our work, we realized that we needed to create a custom parser for a formulaic expression language. We give reasons for this need in subsection 6.4.2. Custom parser implementation is a time-consuming task. We needed a language with an expressivity that is powerful enough to reach the project's goal. Because of time constraints for our work, we had to do a compromise between conciseness of formula language and of easiness of implementation.

We therefore created a language that has a very simple syntax with only few constructs. For a user of the language, the simple syntax can both be considered an advantage and a disadvantage. As an advantage, the simple syntax makes the language arguably easy to learn. As a disadvantage, the lack of concise constructs requires the user to express some aspects more verbose than it might be necessary in other languages.

```
-1234
```

Listing 1 Formula evaluating to an integer value

```
{4.34e1}
```

Listing 2 Formula evaluating to a double value

```
"This is a string"
```

Listing 3 Formula evaluating to a string value

```
true
```

Listing 4 Formula evaluating to a Boolean value

Our language natively supports syntactic constructs to describe integers, doubles, strings and Boolean values. Listing 1 evaluates to the integer -1234, Listing 2 evaluates to the double 43.4, Listing 3 evaluates to the string enclosed within the two quotation marks, and Listing 4 evaluates to the Boolean value for truth.

```
exampleValue
```

Listing 5 Formula consisting of an identifier

Each formulaic expression is evaluated in a so-called “environment”. An environment is a construct that allows mapping identifiers to values. Assuming Listing 5 is evaluated in an environment that maps the identifier `exampleValue` to the integer 3, then the formula evaluates to the integer 3.

Values of additional types other than the presented four ones can be accessed through an environment if provided by it. In some of the following examples, we make use of the generic type called “collection” that allows to represent sequences of values of a single pre-defined type.

To describe the deviation of a new value from some other value, our formulaic language introduces the syntactic construct “accessor”. The notation of accessors in our language is inspired from the dot notation often used by object-oriented programming languages like Java or C#. Our language supports three types of accessors: property accessors, function accessors and lambda accessors.

```
"demo".Length
```

Listing 6 Formula with an application of a property accessor on a string

```
-1234.Negation
```

Listing 7 Formula with an application of a property accessor on an integer

A property accessor yields information that is directly derivable from the base value. For example, appending the property accessor `.Length` to an expression that evaluates to a string

describes the length of that string. As such, Listing 6 evaluates to the integer 4 because the word "demo" consists of four characters. Listing 7 evaluates to negation of the integer -1234, i.e. to the integer 1234.

```
-1234.Plus(234)
```

Listing 8 Formula with an application of a single-argument function accessor on an integer

```
"This is a string".Substring(0, 4)
```

Listing 9 Formula with an application of a double-argument function accessor on a string

A function accessor yields information that can be derived from the base value and from arguments that are provided with the function accessor. For example, appending the accessor `.Plus(2)` to an expression that evaluates to an integer describes the summation of the base value and the integer 2. As such, Listing 8 evaluates to the sum of -1234 and 234, i.e. to the integer -1000. Listing 9 evaluates to the substring consisting of the first four characters of the provided string, i.e. to `This`.

```
exampleCollection.All[ member | member.GreaterThan(3) ]
```

Listing 10 Formula with an application of a lambda accessor

A lambda accessor yields information that can be derived from the base value and from a parametrized formulaic sub-expression that is provided with the lambda accessor. For example, if an expression evaluating to a collection of integers is appended with the accessor `.All[ member | member.GreaterThan(3) ]`, the resulting expression describes the truth of all integers in the collection being greater than three. As such, assuming the environment maps `exampleCollection` to a collection of the integers 5, 4, 3 and 2, then Listing 10 evaluates to $false$ because the collection members 3 and 2 are not greater than three.

```
exampleCollection.All[ item | item.GreaterThan(3) ]
```

Listing 11 Formula with an application of a lambda accessor, using a different parameter name

Parameter names can be chosen freely in lambda accessors (under the syntactic constraints as introduced in subsection 4.3.2), so Listing 10 and Listing 11 are equivalent.

```
"This is a string".Substring(0, 4).Equals("This").Inverse
```

Listing 12 Formula with an accessor chain

Accessors can be chained: Listing 12 evaluates to the inverse of the truth of the first four characters of `This is a string` being equal to `This`, i.e. to false.

```
exampleCollection.First
```

Listing 13 Formula possibly resulting in ***null*** result

Our language uses a notion of nullable types as described in (Wikipedia contributors 2017b) so that each type does not just encompass primitive values but also a special *null* value. This *null* value is typically used to denote an evaluation error. As such, assuming the environment maps `emptyCollection` to an empty integer collection, then Listing 13 evaluates to *null* because there is no first element in an empty collection.

```
emptyCollection.First.Plus(1)
```

Listing 14 Extension of Listing 13 with a following accessor

When applying an accessor to a *null* value, the result will always be a *null* value again. As such, if Listing 13 results in *null*, then Listing 14 evaluates to *null* as well.

Note that even though the language contains "lambda accessor", it is not equivalent to Lambda calculus. In particular, functions are not first-class values in our language. As such, it is not a Turing-complete language.

All available accessors are documented in Appendix C.

4.3.2 Syntax

In this section, we formally introduce the syntax of our formulaic expression language. We specify elements of the syntax as Extended BNF (International Organization for Standardization 1996) syntax rules and informally explain the syntactic constructs.

```
Formula = Base | Formula, ".", Accessor
```

Listing 15 `Formula` syntax rule

The root syntactic element of our language is a `Formula` as specified in Listing 15. A `Formula` either is a `Base` or consists of a dot-prefixed `Accessor` following a `Formula`.

```
Base = Boolean | Integer | String | Double | Identifier
```

Listing 16 `Base` syntax rule

A `Base` allows to instantiate some of the language's primitive types and to refer to an identifier of the environment. As specified in Listing 16, a `Base` is either of `Boolean`, `Integer`, `String`, `Double`, or `Identifier`. We call non-`Identifier` `Base`s "constants".

```
Boolean    = "true" | "false"
Integer    = /([+-]?[0-9]+)/
String     = /("(?:[^"\\]|\\"|\\\\)*")/
```

```
Double     = /({[+-]?[0-9]+\.[0-9]*(e[+-]?[0-9]+)?})/i
Identifier = /([A-Za-z_][A-Za-z_0-9]*)/
```

Listing 17 Syntax rules for `Base`s

The syntax rules for the `Base`s are collectively specified in Listing 17. For the `Base`s different from `Boolean` we deviate slightly from the Extended BNF notation for improved conciseness: In the respective syntax rules we make use of regular expressions as specified in (.NET Docs contributors 2017). To further improve the reader's reading experience, we show these regular expressions with highlighted syntax through colorization.

A `Boolean` is either of the strings `true` or `false`. An `Integer` is a non-empty sequence of numeric digits, optionally prefixed with a plus or a dash sign. A `String` is a sequence of arbitrary characters enclosed in double quotes. If a `"` or `\` character is to be used within the sequence of arbitrary characters of a `String`'s value, it needs to be escaped with a prefix `\`. For the `Double` syntax rule we give an intuitive description: A `Double` equals a curly bracket-enclosed string representation of a double value in a C-like language. An `Identifier` starts with a character of the basic Latin alphabet or an underscore, and continues with a sequence of further basic Latin alphabet characters, underscores, and numeric digits.

```
Accessor = PropertyAccessor | FunctionAccessor | LambdaAccessor
```

Listing 18 `Accessor` syntax rule

As specified in Listing 18, an `Accessor` is either a `PropertyAccessor`, a `FunctionAccessor` or a `LambdaAccessor`.

```
PropertyAccessor = Identifier
```

Listing 19 `PropertyAccessor` syntax rule

Each `PropertyAccessor` has an identifier. To avoid duplicate specifications, the `PropertyAccesor`'s specification as given in Listing 19 reuses the `Identifier` specification.

```
FunctionAccessor = Identifier, "(", ArgumentList, ")"
ArgumentList = Formula | ArgumentList, "," Formula
```

Listing 20 `FunctionAccessor` and its `ArgumentList` syntax rule

A `FunctionAccessor` consists of an identifier followed by a round bracket-enclosed list of arguments. The formal specification is given in Listing 20.

```
LambdaAccessor = Identifier, "[", LambdaParameterList, "|", Formula, "]"
LambdaParameterList = Identifier | LambdaParameterList, ",", Identifier
```

Listing 21 `LambdaAccessor` and its `LambdaParameterList` syntax rule

A `LambdaAccessor` consists of an identifier followed by an opening square bracket, followed by a list of lambda parameters, followed by a vertical bar, followed by a `Formula`, and finally followed by a closing square bracket. The formal specification is given in Listing 21.

4.4 Execution Semantics Description Language

In section 3.1, we have established that business process models can be made accessible for model checking by assigning sequences of behaviors to such models and to model elements. In this section, we introduce the Execution Semantics Description Language (ESDL), a language that we developed to specify execution semantics of a process model in a standardized and automatically processable way.

First, we give explain the design of ESDL. Next, we give an informal introduction into the language and its behaviors. Due to space constraints, we give a formal specification for the language only in Appendix B.

ESDL is based on the concept of assigning sequences of "behaviors" to Models on the level of their modeling Language, and to ElementOccurrences on the level of their Elements' ElementTypes. On this basis, we say that a Model or an ElementOccurrence can be "enabled", thereby triggering the execution of the assigned behaviors. A LTS can be specified based on sequences of outputs of relevant behaviors that can be triggered when enabling some initially selected Model.

We designed our language primarily with usability requirements in mind. The language should be user-friendly in such a way that a human user can easily specify sequences of behaviors and assign them to ElementOccurrences and Models on the ElementType and Langauge level, respectively. Our language is not intended to be written in text. Instead, its behaviors are intended to be put together in a building block-like fashion using a graphical user interface. We therefore do not specify a text-based syntax for ESDL.

In ESDL we assume that enablements of Models and of ElementOccurrences can be "scheduled". If an enablement is scheduled, it is put into a multiset that we call "task list". If execution of a sequence of behaviors completes, an element in the task list will be enabled. The LTS is derived in such a way that all possible execution orders for tasks in task lists are considered.

Behaviors of ESDL allow storing data, accessing stored data, and deleting stored data. Data is stored as instances of CustomStorageData for RuntimeInstances and ElementOccurrences. In this regard, a RuntimeInstance functions as a dictionary that contains mappings from ElementOccurrences to CustomStorageData values.

We define nine behavior types for ESDL, each with their own parameters. For some of these parameters, the respective argument is to be specified as a formula in our formulaic expression language as introduced in section 4.2. For other parameters, the argument is to be specified as sequences of behaviors. Other arguments are to be specified as plain strings or as Booleans.

We introduce these behaviors and their parameters informally in Table 3. In the first column, we give the name of the behavior type. In the second column, we list the parameters for the behavior. For each parameter, we give its name followed by its type. For parameters taking a formulaic expression argument, we write "Formula", followed by an arrow pointing to the data type that the formulaic expression needs to evaluate to. In the third column, we give an information description of the behavior's semantic.

Behavior Type Name	**Parameters**	**Informal Description**
Enable Element Occurrence	Runtime Instance: Formula → $RuntimeInstance$ Element Occurrence: Formula → $ElementOccurence$ Data to pass on: Formula → $CustomEnablementData$ Perform now instead of scheduling it: Boolean	If triggered, the ElementOccurrence will be enabled with the RuntimeInstance and the CustomEnablementData as specified in the respective arguments. If the Boolean flag is set, then the enablement will take place directly; if it is not set, then the enablement will be scheduled.
Enable Model	Create new runtime instance: Boolean Runtime Instance: Formula → $RuntimeInstance$ Model: Formula → $Model$ Data to pass on: Formula → $CustomEnablementData$ Perform now instead of scheduling it: Boolean	If triggered, the respective Model will be enabled with the CustomEnablementData as specified in the respective arguments. If the Boolean flag is set, the enablement will take place directly; if it is not set, the enablement will be scheduled. If the "Create new runtime instance" flag is set, then the enablement will take place with a newly created RuntimeInstance; if it is not set, then the enablement will take place with the RuntimeInstance as specified in the respective argument.
For one item in a collection	Item Variable Name: String Collection: Formula → $Collection\langle T\rangle$ for any allowed Type T Child Behaviors: Sequence of Behaviors	If triggered, the Child Behaviors will be triggered sequentially for some member of the Collection as specified in the respective argument. The LTS will be specified in such a way that all members of the collection will be selected once. Formulas in the Child Behaviors will be evaluated in an environment that maps the Item Variable Name to the selected member and that maps all other identifiers of the current environment to the respective values of the current environment.

Behavior Type Name	**Parameters**	**Informal Description**
For each item in a collection	Item Variable Name: String Collection: Formula → $Collection\langle T\rangle$ for any allowed type T Child Behaviors: Sequence of Behaviors	If triggered, the Child Behaviors will be triggered sequentially, once for each member of the Collection as specified in the respective argument. Each time the Child Behaviors are triggered, Formulas in them will be evaluated in an environment that maps the Item Variable Name to the respective current member and that maps all other identifiers of the current environment to the respective values of the current environment.
If/Then/Else	Condition: Formula → $Boolean$ Then Behaviors: Sequence of Behaviors Else Behaviors: Sequence of Behaviors	If triggered when the Condition evaluates to $true$, the Then Behaviors are triggered sequentially. If triggered when the Condition evaluates to $false$, the Else Behaviors are triggered sequentially.
Load Data	Runtime Instance: Formula → $RuntimeInstance$ Element Occurrence: Formula → $ElementOccurence$ Variable Name: String Child Behaviors: Sequence of Behaviors	If triggered, the Child Behaviors will be triggered sequentially. Formulas in the Child Behaviors will be evaluated in an environment that maps the Variable Name to the CustomStorageData that was stored for the ElementOccurrence and the RuntimeInstance as specified in the respective arguments. If no CustomStorageData had been stored for them before, then the environment will map to the default value for CustomStorageData. All other identifiers of the current environment will be mapped to the respective values of the current environment.
Release Runtime Instance	Runtime Instance: Formula → $RuntimeInstance$	If triggered, all data that was stored for the Runtime Instance will be deleted.
Report Event	Element Occurrence: Formula → $ElementOccurence$ Event Content: Formula → $String$	If triggered, an event will be reported as output. The LTS will be specified based on sequences of this kind of reports. The output is a tuple consisting of the Element Occurrence and the Event Content as specified in the respective arguments.
Store Data	Runtime Instance: Formula → $RuntimeInstance$ Element Occurrence: Formula → $ElementOccurence$ Data to be stored: Formula → $CustomStorageData$	If triggered, the CustomStorageData value as specified in "Data to be stored" will be stored for the tuple of ElementOccurrence and RuntimeInstance as specified in the respective arguments.

Table 3 Informal description of ESDL behavior types

For all formulaic expressions used as arguments in behaviors, a default environment is defined. We list and explain its members in Table 4. The `CurrentLanguageElementTypes` is a special identifier that does not have a true data type in the context of our specification. It allows easy referencing to ElementTypes of the current [em] Language and behaves like an object with two properties: `ObjectTypes` and `RelationshipTypes`. The result of the `ObjectTypes` and the

`RelationshipTypes` properties behave like they have a property for each ObjectType or RelationshipType of the current [em] Language, respectively.

Identifier	Type	Informal Description
`CurrentRuntimeInstance`	*RuntimeInstance*	The RuntimeInstance that the current ElementOccurrence or model was enabled with.
`EnablementData`	*CustomEnablementData*	The data that the current ElementOccurrence or Model was enabled with.
`NewStorageDataInstance`	*CustomStorageData*	The default instance of *CustomStorageData*, i.e. *Default*(*CustomStorageData*).
`NewEnablementDataInstance`	*CustomEnablementData*	The default instance of *CustomEnablementData* , i.e. *Default*(*CustomEnablementData*).
`CurrentLanguageElementTypes`	(special construct)	(See description in main text.)
`CurrentModel`	Model	The currently enabled Model; only available for behaviors assigned to a Model.
`CurrentObjectOccurrence`	ObjectOccurrence	The currently enabled ObjectOccurrence; only available for behaviors assigned to a ObjectOccurrence.
`CurrentRelationshipOccurrence`	RelationshipOccurrence	The currently enabled RelationshipOccurrence; only available for behaviors assigned to a RelationshipOccurrence.

Table 4 Members of the default environment for formulaic expressions

We assume most behavior types to be directly understandable from the descriptions in Table 3, except for the behaviors “For one item in a collection” and “Release Runtime Instance”. We therefore try to facilitate a better understanding for these behaviors and their background by giving further information and intuitions.

We start with an intuition behind the “For one item in a collection” behavior: Assume a process model contains an element with three other following elements, and the process flow may continue from the first element to either one of these three elements. Then it can be desirable that every possible element sequence from the first element to a follower is considered in the derived LTS. This situation is an instance of the Exclusive Choice workflow pattern as specified in (van der Aalst and ter Hofstede 2017, sec. “Pattern 4 (Exclusive Choice)”; van der Aalst et al. 2003).

The behavior “For one item in a collection” is especially helpful for implementing this pattern. In the described situation, a formula specifying the collection of the three followers could be used as the behavior’s Collection argument. Then an “Enable Element Occurrence” behavior with its Element Occurrence argument set to the Item Variable Name could be used as the only child behavior of the “For one item in a collection”. This way, it is ensured that the resulting LTS contains all possible sequences from the first element to a follower.

We continue with an intuition behind the “Release Runtime Instance” behavior. Consider the two BPMN-like process models in Figure 7. The second activity in Process Model I “calls” the Process Model II. Activities in Process Model II stores data and later make use of it. Once Process Model II was processed, execution returns to the third activity in Process Model I. Assume that each of the two Process Models work with their own RuntimeInstance.

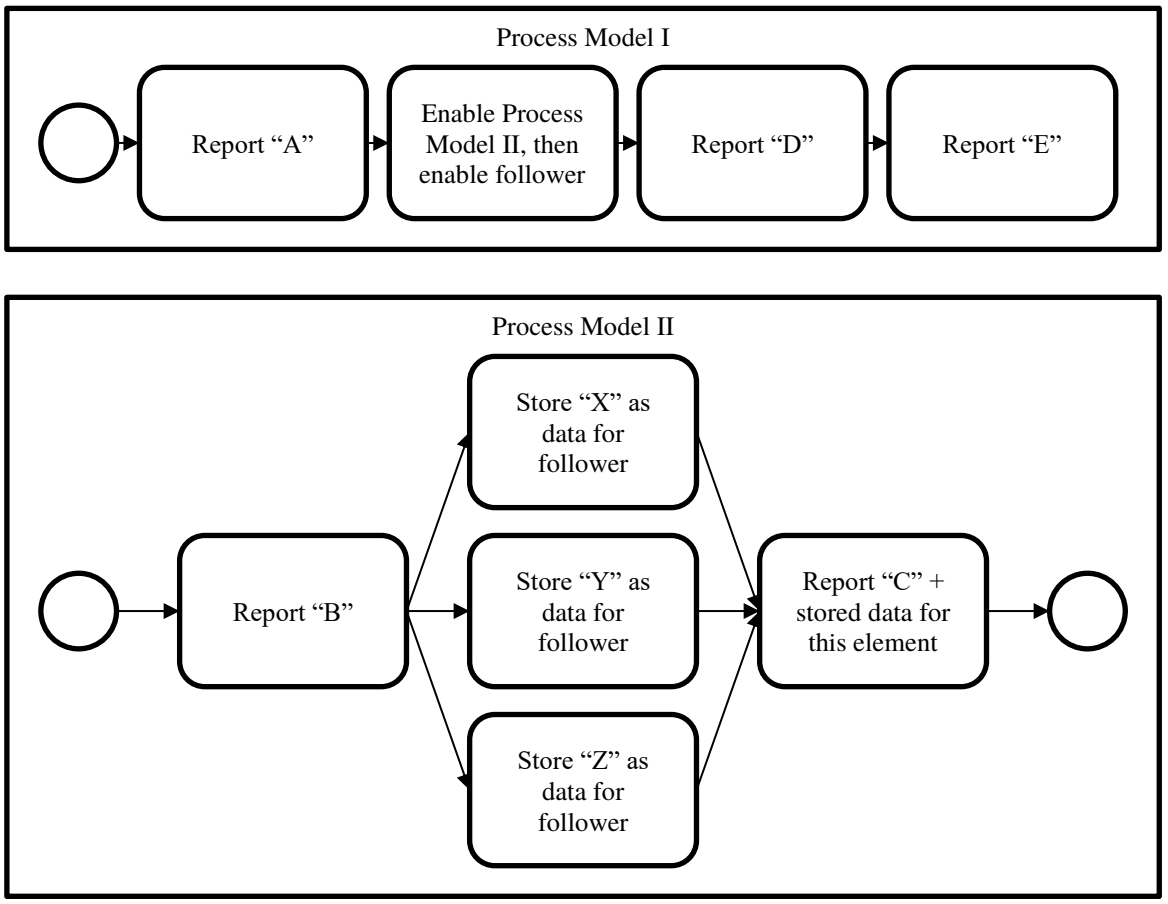

Figure 7 Process models for demonstrating unnecessary LTS size increase that occurs when not releasing unused RuntimeInstances

We can distinguish between two scenarios: In the first scenario, data stored for the RuntimeInstance of Process Model II is not deleted when returning the third activity in Process Model I. In the second scenario, the stored data is deleted when returning. The two different LTS that can be derived from the two scenarios are shown in Figure 8, the first one on the left, the second one on the right. The boxes with the dashed lines indicate in which states the data stored for Process Model II’s RuntimeInstance is kept.

From inspection of the process models in Figure 7 it can be shown that data stored for Process Model II’s RuntimeInstance is not required after execution returns to Process Model I. However, generalizing such an analysis is non-trivial and for some situations possibly impossible. A LTS is therefore derived in a naïve way: Derivation bases on the assumption that data stored for a RuntimeInstance might be used at any point during process model execution.

As such, all states where data is stored for a RuntimeInstance must be kept on their own LTS paths. When data stored for a RuntimeInstance is released however, LTS paths might merge again.

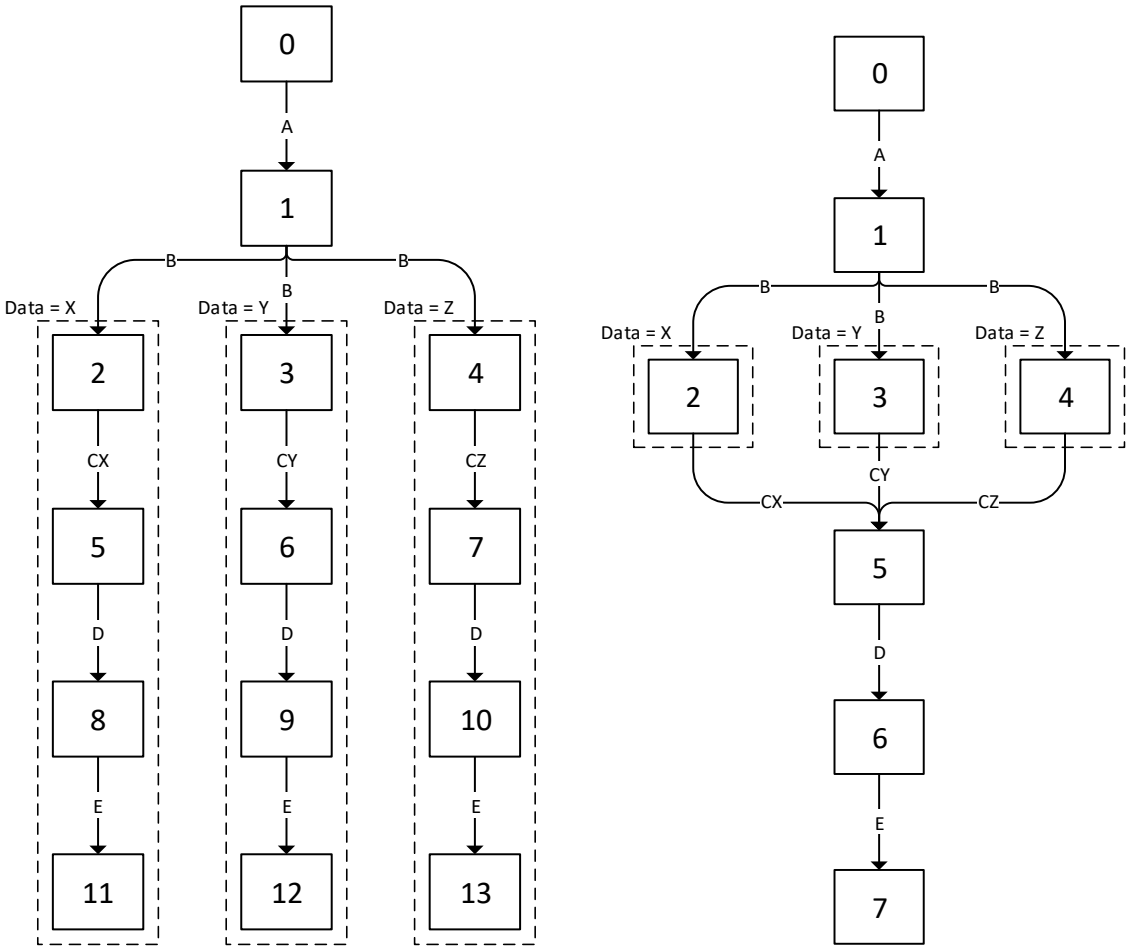

Figure 8 Two LTS that can be derived from the process models in Figure 7

This explains the two different LTS sizes: Comparison of the two LTS shows that keeping data stored for Process Model II's Runtime instance instead of deleting it leads to a larger LTS. Since smaller LTS allow faster processing, it can be considered good practice to add a "Release Runtime Instance" behavior when data is not required anymore that was stored for some RuntimeInstance.

5 Foundations for Implementing the Theoretical Approach

In this chapter, we describe the gathering of information and the development of further concepts that are required to implement our abstract approach. While the two previous chapters remained mostly on the theoretical level, this chapter considers primarily practical problems.

In the first section, we describe how we searched for model checkers and why we picked the *Construction and Analysis of Distributed Processes* (CADP) model checker for our implementation. In the second section, we introduce our approach for translating [em] data and assigned behavior sequences into a formal process specification that can be processed by the CADP model checker. In the third section, we introduce a CADP-supported temporal property specification language modification that extends the language with macros as described in subsection 3.2.2. In the fourth section, we collect requirements that an implementation of our approach must fulfill to be usable for a user.

5.1 Searching and Selecting a Suitable Model Checker

An implementation of the theoretical approach requires a component that performs model checking based on the user's inputs. Such a component could either be implemented from scratch or an existing model checker could be integrated into the final implementation.

To avoid the effort of implementing a complete model checker, we decided to use an existing one for our implementation. In this section, we describe how we searched a suitable model checker and why we selected the CADP model checker as the basis for our implementation.

In the first subsection, we describe the requirements we laid down for a model checker to be used by our implementation. In the second subsection, we describe the approach we took to select a suitable model checker.

5.1.1 Model Checker Requirements

In this subsection, we introduce our model checker requirements (RM).

In section 2.2, we have established that temporal property specification languages vary w.r.t. their expressivity. Less expressive languages cannot describe temporal properties of a certain complexity. We assume that the usability of an implementation of our approach is higher if it enables users to check their models for the fulfillment of more complex temporal properties. From this assumption, we derive the first model checker requirement:

RM 1. The model checker must accept temporal properties that are formulated in an expressive temporal property specification language.

In subsection 5.1.2, we go into more detail on what we consider an expressive temporal specification language.

In subsection 3.2.2, we have established that a temporal property description language must support placeholders. This leads to the second model checker requirement:

RM 2. It must be possible to introduce placeholders in temporal properties as described in subsection 3.2.2 in a way that the model checker accepts these properties.

As established in section 3.2, our approach relies on the availability of counterexample or witness graphs. This leads to another model checker requirement:

RM 3. The model checker must return a counterexample or witness graph as part of its model checking result that allows inference of result-explaining model element sequences as described in section 3.2.

In section 3.3, we have established that a transformer is required to transform models, information about modeling languages used by the models, and execution semantics defined for these languages, into a formal process described in a formal process specification language that the model checker accepts. Many model checkers only accept formal processes specified in their respective custom process specification languages and custom file formats. Some of these languages and file formats are complex and make generation of the required inputs difficult. To complete the work around this thesis within the allocated timeframe, we established as a requirement:

RM 4. The model checker must accept formal processes that are specified in a process specification language and in a file format that is designed in such a way that developing and implementing a transformer as described in section 3.3 is easily possible.

Model checkers differ with regards to the systems and platforms they can be run on. If a model checker is to be controlled from a tool running on a system that is not supported by the model checker, a cross-system or cross-platform communication protocol between the controlling tool and the model checker must be developed. In section 8.2.1, we discuss how such a communication protocol might make computations faster in future implementations. For our work, we decided to save the effort of developing such a protocol. We therefore established as a requirement:

RM 5. The model checker must be runnable on the host system, i.e. the system that the tool runs on that controls the model checker.

During initial research we found several model checkers that were not maintained anymore and could not be run on modern machines without investing additional work. Further we assumed

that it would be helpful to get support from the model checker developers or maintainers when integrating the selected model checker into an implementation and when generating the first model checking problems to be solved by it. During our evaluation of model checkers, we consequently established as a soft requirement:

RM 6. The model checker should be actively maintained.

5.1.2 Surveying Model Checkers

In this section, we describe our process of finding model checker candidates that we considered to use in our implementation, and of selecting a suitable model checker from the candidates.

To find model checker candidates, we performed an internet search. We used the Google search engine with the terms “model checker” and “model checking tool”. Initial searches brought us to a list of model checking tools on the English Wikipedia (Wikipedia contributors 2017a). In further searches, we did not find any other tools fulfilling our requirements and not being listed on the Wikipedia page. We therefore used the entries on the Wikipedia page as our main source for candidate model checkers.

We evaluated the found candidates w.r.t. fulfillment of our requirements as described as follows.

To determine if a model checker supports expressive temporal property specification languages, we checked if μ-calculus or similar expressive derivatives from μ-calculus (like alternation free μ-calculus) were amongst its supported algebras. To determine if a model checker supports placeholders and if it could generate counterexample or witness information as one of its results, we checked its documentation.

We also checked the model checker’s documentation to evaluate the formal process specification languages and file formats it accepts w.r.t. simplicity of implementing a transformer as described in section 3.3. We assumed that integration of model checkers would be difficult if they used binary formats, or formats based on graphical formal process specification languages. We assumed that text-based specification languages and file format would make our implementation simpler.

To determine a model checker’s degree of maintenance, we checked the age of the latest released version of the respective tool and the age of the newest entries on the maintainer’s official communication channels like mailing lists or forums.

In our evaluation we identified two candidates that we found most interesting: The *micro Common Representation Language 2* (mCRL2) toolset and the *Construction and Analysis of Distributed Processes* (CADP) software tools. Being Unix-targeted tools, the CADP tools can

be run on Windows only in a non-native environment (e.g. using Cygwin or a virtualized Linux operating system). Running them on Windows in such an environment makes them operate slowly. The mCRL2 toolset on the other hand has the advantage of running natively on Windows without negative impacts on model checking speed.

In the end, we still had to abandon mCRL2 as a candidate because the counterexample and witness information generated by it were provided in a way that did not allow inference of result-explaining model element sequences as described in section 3.2. Consequently, we selected the CADP model checker as the foundation of our implementation. We still consider mCRL2 an interesting candidate and further discuss its potential applicability for future implementations in section 8.2.

5.2 Generating Formal Processes from [em] Data and Behaviors

As established in section 3.3, an [em] Model and further relevant information must first be tranformed into a formal process specification before it can be processed by a model checker.

CADP primarily supports the formal process specification languages LOTOS and LNT. LOTOS is a ISO-standardized formal process specification language to describe communication protocols and distributed systems. (Bolognesi and Brinksma 1987) LOTOS is a complicated language and it can be difficult and laborious to write or generate formal process specifications in it. To make specifying LTS models less tedious, LNT was developed as a replacement for LOTOS that is equally expressive as LOTOS but easier to use. (Champelovier et al. 2017) Being the formal process specification language that was easier to generate code in, we picked LNT as our translation target language.

This section contains some LNT code. LNT resembles a procedural programming language like Pascal. For brevity, we do not give a detailed introduction into LNT. An interested reader is referred to (Champelovier et al. 2017) for a detailed reference.

We developed an approach to generate a formal process specification in LNT from a set of [em] projects, from custom types specifications for our formulaic expression Language, and from sequences of behaviors that are assigned to Models on the level of their modeling Language, and to ElementOccurrences on the level of their Elements' ElementTypes. We present the detailed LNT generation approach in the following. A visualization of its workflow is given in Figure 9 that follows our notation as introduced in section 3.3.

The formal process specification is generated as multiple units that are finally assembled as one LNT specification.

The unit "static code" contains foundational LNT specifications that remaining code builds upon. It especially contains an LNT implementation of nullable types, of a LNT `process` to

launch a Model's sequence of behaviors, of another LNT `process` to launch an ElementOccurrence's sequence of behaviors, and of LNT types and functions required for data storage management as well as for scheduling.

The unit "[em] data" contains a LNT representation of the given [em] Projects with their Models and the Languages they use. We explain how the translation of [em] data works and how the generated unit is built up in subsection 5.2.1.

The unit "entry points for [em] Models" contains a LNT root `process` for each given [em] Model. Such a root `process` can be considered a model checking entry point for the respective [em] Model and corresponds to the initial state of a formal process entailed by it.

The unit "custom types" contains a LNT representation for the custom types that were specified for our formulaic expression language.

The unit "behaviors" contains the LNT representation of the sequences of behaviors specified for the ElementOccurrences and Models. As behaviors contain formulaic expressions, translating behaviors also involves translating formulaic expressions. We explain how the translation of formulaic expression works and how the resulting LNT code is built up in subsection 5.2.2. On this basis, we explain the same for the translation of sequences of behaviors in subsection 5.2.3.

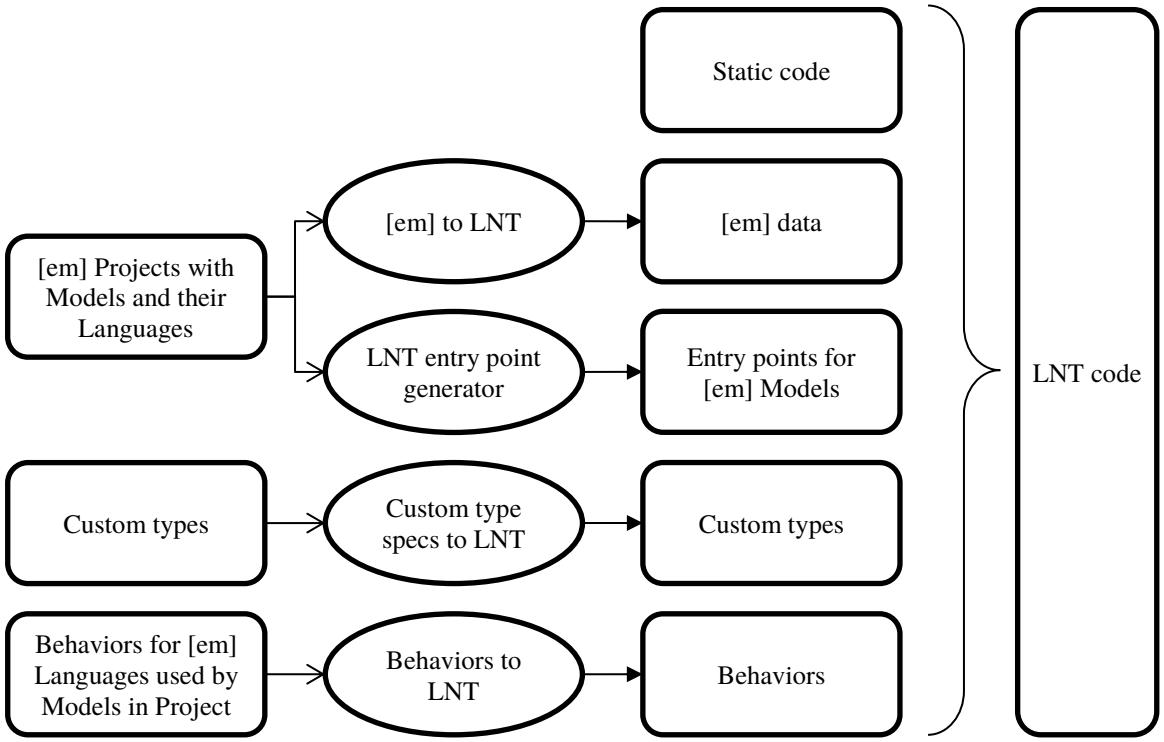

Figure 9 LNT generation workflow

5.2.1 Translation of [em] Data to LNT

A set of [em] Projects is required as input for the translation of [em] data to LNT. From this input, a set of Elements and a set of Models in these Projects can be derived. These sets allow deriving remaining required information, especially the set of Languages used by the Models and the set of ElementOccurrences in the Models.

Based on the input and its derived information, the five chunks of LNT code in Table 5 are to be generated for each class of [em] data model as introduced in section 4.1.

Chunk 1. A LNT type declaration corresponding to the [em] data model class.

```
type TYPE_ID_EM_OBJECT is
    NULL_VAL !implementedby "NULL_VAL_TYPE_ID_EM_OBJECT",
    ID_EM_OBJECT_26738691,
    ID_EM_OBJECT_26738706,
    ID_EM_OBJECT_26738697
    with "==", "!=", "<"
end type
```

Listing 22 Exemplary LNT type declaration for representing [em] Objects

For each instance of the class, the LNT type corresponding to it must contain a parameter-less constructor that represents the respective instance. An additional constructor is required represent *null*. Our LNT code requires each constructor identifier to be unique. Introducing the name of the class and [em]'s instance ID into the identifier ensures fulfilment of this requirement. We give an example of an LNT type declaration for the [em] Object class with three exemplary instances in Listing 22.

Chunk 2. A LNT type declaration corresponding to a nullable list of instances of the class.

```
type TYPE_IDLIST_EM_OBJECT is
    list of TYPE_ID_EM_OBJECT
    with "head", "tail", "length", "append", "union", "empty"
end type
type TYPE_NULLABLE_IDLIST_EM_OBJECT is
    NULL_VAL !implementedby "NULL_VAL_TYPE_NULLABLE_IDLIST_EM_OBJECT",
    THE_VAL(VALUE : TYPE_IDLIST_EM_OBJECT)
    with "get"
end type
```

Listing 23 LNT type declaration for a nullable list of instances of the [em] Object class

Specifications for a list of instances for some [em] class are represented as two LNT types: A native LNT list type for instances of the [em] class, and a nullable type that allows storing an instance of the native LNT list type. We give the LNT type specifications for a list of instances of the [em] Object class in Listing 23.

Chunk 3. A function that yields a list of all instances of the respective class.

```
function GET_ALL_EM_OBJECT_IDS : TYPE_NULLABLE_IDLIST_EM_OBJECT is
    return THE_VAL({
        ID_EM_OBJECT_26738691,
        ID_EM_OBJECT_26738706,
        ID_EM_OBJECT_26738697
    })
end function
```

Listing 24 LNT function yielding an exemplary list of all [em] Object class instances

We give the function yielding a list of all three Object instances of our previous example in Listing 24. Note that LNT uses curly braces to specify instances of list types.

Chunk 4. For each attribute of the class: An "attribute getter function" that allows retrieving the value of the attribute. We consider a relationship member end to also be an attribute if its target multiplicity is at most one, i.e. if it links to at most one associated target instance.

```
function GET_CAPTION(ID : TYPE_ID_EM_OBJECT) : NULLABLE_STRING is
    case ID in
        ID_EM_OBJECT_26738691 -> return THE_VAL("Open file")
      | ID_EM_OBJECT_26738706 -> return THE_VAL("Do work")
      | ID_EM_OBJECT_26738697 -> return THE_VAL("Close file")
      | any -> return NULL_VAL
    end case
end function
```

Listing 25 Exemplary LNT function yielding the value of an Object's Caption attribute

Each of the getter functions accepts an instance of the LNT type corresponding to respective class. Its implementation contains a case-based control mechanism that returns the value of the attribute of the [em] class instance for the constructor that it represents. We give an example of an attribute getter function that yields the Caption attribute's value of a given [em] Object in Listing 25.

Chunk 5. For each member end of an association of the class with a target multiplicity greater than one: A function that that allows retrieving a list of instances that are linked via the respective association to a given instance of the respective class.

This is implemented using two types of functions: a parameter-less helper function for each of the respective class' instances and a non-helper getter function taking an instance of the respective argument.

```
function GET_EM_OBJECT_26738691_RELATIONSHIPS_WITH_ME_AS_SOURCE_MEMBER_IDS :
TYPE_NULLABLE_IDLIST_EM_RELATIONSHIP is
    return THE_VAL({
        ID_EM_RELATIONSHIP_26738723
    })
end function

function GET_EM_OBJECT_26738706_RELATIONSHIPS_WITH_ME_AS_SOURCE_MEMBER_IDS :
TYPE_NULLABLE_IDLIST_EM_RELATIONSHIP is
    return THE_VAL({
        ID_EM_RELATIONSHIP_26738726
    })
end function

function GET_EM_OBJECT_26738697_RELATIONSHIPS_WITH_ME_AS_SOURCE_MEMBER_IDS :
TYPE_NULLABLE_IDLIST_EM_RELATIONSHIP is
    return THE_VAL({
    })
end function
```

Listing 26 Exemplary LNT helper functions yielding instances linked via the "Followers" association for three [em] Objects

We give such exemplary helper functions for our previous example's Objects in Listing 26.

```
function GET_RELATIONSHIPS_WITH_ME_AS_SOURCE(ID : TYPE_ID_EM_OBJECT) :
TYPE_NULLABLE_IDLIST_EM_RELATIONSHIP is
    case ID in
        ID_EM_OBJECT_26738691 -> return
            GET_EM_OBJECT_26738691_RELATIONSHIPS_WITH_ME_AS_SOURCE_MEMBER_IDS
      | ID_EM_OBJECT_26738706 -> return
            GET_EM_OBJECT_26738706_RELATIONSHIPS_WITH_ME_AS_SOURCE_MEMBER_IDS
      | ID_EM_OBJECT_26738697 -> return
            GET_EM_OBJECT_26738697_RELATIONSHIPS_WITH_ME_AS_SOURCE_MEMBER_IDS
      | any ->
            return NULL_VAL
    end case
end function
```

Listing 27 Exemplary LNT getter function yielding instances linked via the "Followers" association for [em] Objects

The non-helper getter function yields the result of the correct helper function, using a case-based control mechanism like introduced above in the context of attributes. We give an example of such a getter function in Listing 27.

Table 5 Chunks of LNT code to be generated for [em] classes and their instances

Some classes of the [em] data model have an inheritance hierarchy. While not presented here in detail for brevity, our generated LNT code contains additional concepts to convert between the different inheritance levels of these classes.

5.2.2 Translation of Formulas to LNT

Several behavior types have formulaic expression parameters. When behaviors of these types are to be transformed to LNT, their formulaic expressions must be translated as well. In this subsection, we present our translation approach.

As first step, the given formulaic expression must be parsed, yielding an Abstract Syntax Tree (AST) that corresponds to the expression. The AST must then be traversed to generate the required outputs. The primary output is an LNT expression that can be inserted into a LNT `process`. A secondary output is a set of LNT helper `function`s that need to be referred to in the primary output's LNT expression. We describe how the AST needs to be traversed to generate both the LNT expression and the code of the additionally required helper `function`s.

A mapping of identifiers to LNT expressions is kept in memory during AST traversal. Initially, this mapping contains LNT expressions that correspond to the entries of the environment in which the formula is evaluated. This mapping is extended as required, especially when generating code for the helper `function`s.

If a constant node is visited during tree traversal, a LNT representation of the constant is added to the primary output. If an Identifier node is visited during tree traversal, an expression is added to the primary output according to the mapping of identifiers as described above.

If a `PropertyAccessor` node, a `FunctionAccessor` node, or a `LambdaAccessor` node is visited during tree traversal, an LNT expression is added to the primary output that invokes a LNT function yielding a result according to the accessor's specification. For all three accessor types, the respective accessor's `Base` is visited to generate the LNT code for one of the arguments of the LNT function invocation.

For `FunctionAccessor` nodes, also the `Argument` nodes are visited to generate LNT code for further arguments of the LNT function invocation. For `LambdaAccessor` nodes, a recursive helper function is added to the secondary output and an invocation of this function is added to the primary output. The helper `function` contains the LNT translation result of visiting the accessor's sub-`Formula`.

```
CurrentObjectOccurrence.RelationshipOccurrencesWithMeAsSource.Count.GreaterThan(0)
```

Listing 28 Formulaic expression to determine the existence of outgoing Relationships from the current ObjectOccurrence

To exemplarily demonstrate results of our approach of translating formulaic expressions to LNT code, we give a typical expression, then show the LNT code that it translates to, and explain relevant aspects of the result. The formulaic expression for our demonstration is given in Listing 28.

```
(custom_count(OBJECT_OCCURRENCE_ID.RELATIONSHIP_OCCURRENCES_WITH_ME_AS_SOURCE)) >
(THE_VAL(+0))
```

Listing 29 LNT expression corresponding to the formulaic expression in Listing 28

Assuming the identifier `CurrentObjectOccurrence` maps to the LNT expression `OBJECT_OCCURRENCE_ID`, we give the translation of the exemplary formulaic expression into LNT in Listing 29.

The identifier `CurrentObjectOccurrence` translates directly into the mapping's LNT expression. The `RelationshipOccurrencesWithMeAsSource` property accessor translates into a LNT getter function call using dot notation.

```
function custom_count(VAL : TYPE_NULLABLE_IDLIST_EM_RELATIONSHIP_OCCURRENCE) :
NULLABLE_INT is
    case VAL in
        NULL_VAL ->
           return NULL_VAL
      | any ->
           return THE_VAL(NatToInt(length(VAL.VALUE)))
    end case
end function
```

Listing 30 `custom_count` function for a list of RelationshipOccurrences

For the `Count` property accessor working on lists of RelationshipOccurrences, a helper function `custom_count` is required that yields a *null* output for a *null* input and the number of items in the collection otherwise. The LNT code for this helper function is given in Listing 30. It would be possible to include a function of this kind in the "static code" unit for each class of the [em] data model. However, LNT code processing time can be saved when generating this function only if some formula requires it for some class. It therefore makes sense to let this function become one of the secondary output's helper functions.

```
function _>_(VAL1 : NULLABLE_INT, VAL2 : NULLABLE_INT) : NULLABLE_BOOL is
    case VAL1 in
        NULL_VAL -> return NULL_VAL
      | any ->
            case VAL2 in
                NULL_VAL -> return NULL_VAL
              | any -> return THE_VAL(VAL1.VALUE > VAL2.VALUE)
            end case
    end case
end function
```

Listing 31 `>` function for two nullable integers

With `custom_count` generated as a helper function, the `Count` property accessor translates into a call to this function. A helper function `>` that compares two nullable integers is required for the `GreaterThan` function accessor, similar to the `custom_count` function introduced above. We

give this helper function in Listing 31. With the helper function `>` generated, the `GreaterThan` function accessor translates into a call to this function using infix notation.

The constant 0 translates into the corresponding instantiation of the nullable data type with the respective value. The `+` sign is required in LNT to indicate that the number should be interpreted as an integer instead of a natural number. Surrounding a function name with two underscores in LNT makes the function an infix function, i.e. it can be invoked using infix notation.

Some helper function may be required by multiple formulaic expressions. If such a function was generated multiple times, the required time to process the LNT files processing would be longer. To avoid long processing times, we recommend keeping track over the generated helper function and add such a function to the secondary output only if it had not been generated before. To apply this idea to the given example: If further translations of LNT formulas require any of the functions `custom_count` and `>`, neither of them should be generated a second time. Instead, the existing helper functions should be re-used.

5.2.3 Translation of Behaviors to LNT

The translation of behavior sequences assigned to Models and ElementOccurrences on the meta-level comprises the core of the LNT code specifying formal processes that can be checked with CADP's model checker. We describe how the translation works in our approach.

Our approach assumes that behaviors are available as a tree-like structure: To an implementation of our approach, a behavior containing a sequence of other behaviors needs to be available as a node with a child node for the sequence of other behaviors. Like in our approach of translating formulas to LNT, the tree-like behavior structure needs to be traversed to generate the LNT code that corresponds to the behavior sequences. The primary output of our translation is a sequence of LNT statements that can be inserted into a LNT `process`. A secondary output is LNT helper `process`es and `function`s that need to be referred to in the primary output expression.

Many behaviors can be translated to LNT in a straightforward way. For some behaviors, more complicated translations are required. We explain notable aspects of our approach's translations for different behavior types in Table 6.

If/Then/Else: If a "If/Then/Else" node is visited, the "Condition" argument's formulaic expression needs to be translated to LNT and the behaviors in the sequences provided in the other two arguments "Then Behaviors" and "Else Behaviors" need to be visited for translation to LNT. The resulting LNT code pieces are to be assembled in a suitable LNT conditional statement like `if` or `case`. This statement finally needs to be added to the primary output.

Enable Element Occurrence: If a "Enable Element Occurrence" node is visited, the formulaic expressions in the arguments "Runtime Instance", "Element Occurrence", and "Data to pass on" need to be translated to LNT. The resulting expressions need to be assembled in a LNT statement. The way how the expressions are assembled depends on the value of the "Perform now instead of scheduling it" argument. If this argument is $true$, the statement must instantiate the "static code" unit's LNT `process` that launches a given ElementOccurrence's sequence of behaviors. If the argument is $false$, the statement must schedule it, i.e. to add a task to the task list to launch the ElementOccurrence's sequence of behaviors later.

Report Event: A "Report Event" behavior must introduce the respective event into the LTS that is entailed by the process model. We use LNT's concept of "gates" to introduce events into the LTS. For brevity, we do not go into detail how this concept works but only give a rough description of how event reporting is realized in our approach. An interested reader is referred to (Bolognesi and Brinksma 1987) for an introduction into the concept of gates and to (Champelovier et al. 2017) how LNT implements this concept.

Each of the LNT root `process`es in the unit "Entry points for [em] Models" needs to contain a declaration of a gate that is used for reporting events. In our implementation, we named this gate `REPORT_EVENT`. All other LNT `process`es that potentially report events need to be synchronized with this gate. If any such `process` offers a communication label via `REPORT_EVENT`, the LTS entailed by the overall formal process contains a transition with a label that corresponds to the gate name and the offered communication label.

In our implementation, we established a convention for communication labels that implements the concept of "private" events as introduced in subsection 3.2.2: By our convention, each communication label offered via `REPORT_EVENT` needs to be a pair of

1. an LNT constructor that corresponds to the ElementOccurrence as specified in the formulaic expression of a behavior's "Element Occurrence" argument, and

2. a nullable string corresponding to this behavior's Event Content argument.

When visiting a "Report Event" behavior during tree traversal, the formulaic expressions in the arguments "Element Occurrence" and "Content" need to be translated to LNT. The two resulting translations must become the "arguments" of a LNT statement that offers the arguments on the `REPORT_EVENT` gate. This statement needs to be added to the primary output.

For one item in a collection: There are different ways to implement the "For one item in a collection" behavior in LNT. As one approach, LNT's "non-deterministic assignment" (using the `any` wildcard construct) can be used to select any number smaller than the collection's

length that then serves as an index to retrieve the respective collection's member at the index. As another approach, recursive helper `process`es can be implemented that iterate through the collections, offering at each item if the current item or some next item should be selected. We established the advantages and disadvantages in the context of (Pribnow 2016a). Our current implementation uses non-deterministic assignments because it is easier to implement and it results in a smaller state space during evaluation, thereby increasing evaluation speed.
For each item in a collection: When visiting a "For each item in a collection" behavior during tree traversal, a recursive helper `process` needs to be generated that iterates through the collection. This helper process must contain the LNT translation of the behavior's Child Behaviors. A statement invocating this helper `process` needs to be added to the primary output.

Table 6 Notable aspects on the translation of behaviors to LNT

A behavior's argument might contain a formulaic expression resulting in a value of a type that does not match the type that is expected for the behavior's argument. In such a case, the generated LNT code might be invalid and could lead to later error messages that are hard to understand. To find such errors early and to be able issue understandable error message, we recommend checking the result types of formulaic expressions in behavior arguments during translation of behaviors to LNT. We further recommend stopping translation of behaviors if a formula resulting in an invalid type is found. Our implementation follows this recommendation.

During model checking, a formulaic expression might evaluate to *null*. As handling of null is not defined for most behaviors, we recommend adding for each formulaic expression LNT statements that check if the expression evaluates to *null*. If it does, the statements should indicate such a situation as an error in a suitable way. Again, our implementation follows this recommendation.

To demonstrate our approach for translating behaviors to LNT, we give an example behavior specification and show its LNT translation.

The exemplary sequence of behaviors is given in Figure 10. It contains a "Report Event" behavior, followed by a "If/Then/Else" behavior that contains an "Enable Element Occurrence" behavior in its "Then Behaviors" argument.

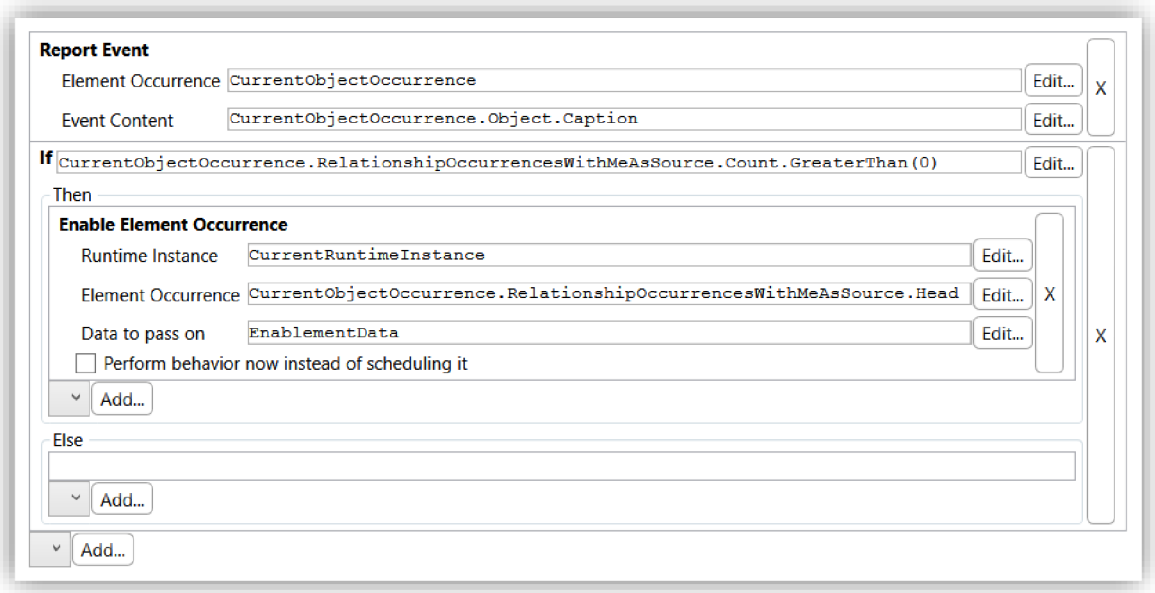

Figure 10 A simple sequence of behaviors as displayed by our implementation

The LNT code for the example's "Report Event" behavior is given in Listing 32. The code declares variables for capturing the results of the formulaic expressions in the behavior's arguments. It assigns the results of these formulaic expressions to the variables. The right-hand side of each assignment statement is the translation of the respective formulaic expression to LNT. Next, the code contains a check of the variables for a *null* value. If any of them has a *null* value, an error is reported and further processing stops. Otherwise, the values of the variables are offered on the `REPORT_EVENT` gate.

```
var elementOccurrence_1 : TYPE_ID_EM_ELEMENT_OCCURRENCE,
    theContent_1 : NULLABLE_STRING in
        elementOccurrence_1 := OBJECT_OCCURRENCE_ID.AS_ELEMENT_OCCURRENCE;
        theContent_1 := OBJECT_OCCURRENCE_ID.OBJECT.CAPTION;
        if((elementOccurrence_1 == NULL_VAL) or_else
           (theContent_1 == NULL_VAL)) then
                ERROR("A null value was provided to a Report Event behavior.");
                stop
        end if;
        REPORT_EVENT(elementOccurrence_1, theContent_1)
end var
```

Listing 32 LNT translation of the "Report Event" behavior of Figure 10

The LNT code for the example's "If/Then/Else" behavior is given in Listing 33. The code is based on a case-based control mechanism that checks the result of the formulaic expression in the behavior's "Condition" argument. The control expression of the case statement is the translation of the respective formulaic expression to LNT. If the formulaic expression results in *null*, an error is reported and further processing stops. If it results in $false$, nothing is done due to the Else Behaviors being an empty sequence. If it results in $true$, the LNT translation of the Then Behaviors will be run. In Listing 33, we indicate the LNT translation of the Then

Behaviors with the placeholder ***ThenBehaviors***. In our example, the Then Behaviors is a sequence containing a single "Enable Element Occurrence" behavior.

```
var condition0 : NULLABLE_BOOL in
      condition0 := ((custom_count(OBJECT_OCCURRENCE_ID.
                     RELATIONSHIP_OCCURRENCES_WITH_ME_AS_SOURCE)) > (THE_VAL(+0)));
      if (condition0 == NULL_VAL) then
            ERROR("The if formula of an if behavior resulted in NULL."); stop
      elsif (condition0 == THE_VAL(TRUE)) then
            ThenBehaviors
      end if
end var
```

Listing 33 LNT translation of the "If/Then/Else" behavior of Figure 10 with a placeholder for the LNT translation of its Then Behaviors

The LNT code for the example's "Enable Element Occurrence" behavior is given in Listing 34. Like in the "Report Event" translation, variables for the two behavior arguments are declared, assigned with the results of the respective formulaic expressions, and checked for null. If no null value is found, the list in the variable `task_list` will be extended with a new item that corresponds to a task of enabling the given ElementOccurrence with the given Event Content. The variable `task_list` maintains the scheduler's list of tasks. It is available to all LNT `process`es generated from behavior specifications.

```
var RuntimeInstance_3 : RUNTIME_INSTANCE_TYPE,
    elementOccurrence_3 : TYPE_ID_EM_ELEMENT_OCCURRENCE in
        RuntimeInstance_3 := INSTANCE_NUMBER;
        elementOccurrence_3 := custom_head(OBJECT_OCCURRENCE_ID.
                RELATIONSHIP_OCCURRENCES_WITH_ME_AS_SOURCE).AS_ELEMENT_OCCURRENCE;
        if((RuntimeInstance_3 == NULL_VAL) or_else
           (elementOccurrence_3 == NULL_VAL)) then
                ERROR("A null value was provided to a Enable Element Occurrence behavior.");
                stop
        end if;
        task_list := append(ELEMENT_OCCURRENCE_TASK(RuntimeInstance_3,
                             elementOccurrence_3, ENABLEMENT_DATA), task_list)
end var
```

Listing 34 LNT translation of the "Enable Element Occurrence" behavior of Figure 10

5.3 Making the Model Checker's Property Specification Language Support Macros

In subsection 3.2.2, we established that it may be helpful for a temporal property specification language to support "macro" that allow easy element-independent event specifications. In this section, we introduce a modification of the CADP-supported temporal property specification language. Our modification extends the language with the described macros.

In the first section, we present our macro extension from a user perspective. In the second section, we show how we internally expand macros written with our extension.

5.3.1 Our Macro Extension for the Property Specification Language MCL

CADP's supported temporal property description language is MCL. (CADP manual authors 2017f; Mateescu and Thivolle 2008) Based on the ideas described in subsection 3.2.2, we wanted our plugin's users to work with a temporal property specification language that supports macros for rewriting public events into their private equivalents. As such, we defined a slightly extended version of MCL that introduces the wanted macro support. For realizing this macro support, our MCL extension introduces a pattern that can easily be found and replaced using a regular expression and replacement of some special characters.

To refer to a public event in our MCL extension, a user surrounds the relevant event content in a triple of curly braces. For example, if a temporal property should refer to a public event with the content `Some Event`, then the public event can be specified as `{{{Some Event}}}`. We wanted to avoid our pattern conflicting with the syntax of plain MCL. We therefore designed out pattern so that it was unlikely that it would need to appear in a temporal property. Since MCL does not have any syntactic construct involving three curly braces, we considered our pattern to be "safe enough" in this regard.

This design choice results in public events with an Event Content that contains three closing curly braces not being specifiable with a macro as described here. We assume however that, in practice, Event Contents do not contain three closing curly braces. Therefore, we do not consider this restriction to be relevant for practical use. And even if Event Contents with three curly braces were relevant, there would be alternative ways to handle Event Contents with three curly braces. For example, a user could adjust the behavior specifications to escape character sequences of three closing curly braces in Event Contents. Alternatively, a user could write a relevant temporal property directly in the expanded version, i.e. without using macros.

```
\{\{\{((?:[^}]|\}(?!\}\}))*)\}\}\}
```

Listing 35 Regular expression to capture a macro as introduced by our MCL extension

The pattern for a macro as introduced here is given in Listing 35 as a regular expression according to the specification in (.NET Docs contributors 2017). To further improve the reader's reading experience, we show the regular expression with highlighted syntax through colorization. The pattern matches a string consisting of 1) three opening curly braces, followed by 2) a sequence of characters that are each either a) no closing curly brace or b) a closing curly brace that is not followed by two additional closing curly braces, and finally ended with 3) three closing curly braces. The macro's Event Content is captured as a "regular expression subgroup" between the initial three opening curly braces and the three closing curly braces. A captured subgroup is used when expanding the macro.

5.3.2 Translating from Macro-Extended MCL to Plain MCL

CADP's model checker requires an input temporal property to be specified in plain MCL. A temporal property that is specified in our extended version of MCL therefore needs to be converted into plain MCL before it can be used for model checking with CADP.

We convert from extended MCL to plain MCL by "expanding" macros. Expansion means to replace each macro instance with a plain MCL construct that matches the given public event. In subsection 5.2.3, we have introduced our approach of using a gate `REPORT_EVENT` to entail an event as a transition in the modeled LTS. So, when using our approach, the MCL construct we want our macros to substitute should match a suitable transition label generated through a communication offer via `REPORT_EVENT`.

CADP offers a text-based encoding for transition labels. In we Listing 36 give a template of a transition label's encoding in an LTS entailed by a formal process specified as LNT code that is generated by our implementation. In this template, ***`ElementOccurrenceConstructor`*** and ***`EncodedEventContent`*** serve as placeholders for the identifier of the LNT constructor corresponding to an ElementOccurrence and for the encoded value of the nullable string corresponding to an Event Content of some "Report Event" behavior, respectively.

```
REPORT_EVENT !ElementOccurrenceConstructor !THE_VAL (EncodedEventContent)
```

Listing 36 Template of CADP's string encoding of a LTS transition label entailed by our implementation's LNT code

Thanks to this text-based encoding, we can leverage MCL's regular expression feature for matching suitable communication labels. When enclosing a string in single quotes (') in MCL, the quote-enclosed string is interpreted as a regular expression as specified in (CADP manual authors 2017h) for matching a communication label. (CADP manual authors 2017f) Note that the regular expressions used in MCL that are specified in (CADP manual authors 2017h) are significantly different from the ones used by C#/.NET as specified in (.NET Docs contributors 2017).

```
 ('REPORT_EVENT !.* !THE_VAL (PreparedEventContent)')
```

Listing 37 Substitute template for occurrences of the macro pattern

Following this idea, we replace each pattern instance with the plain MCL construct given in Listing 37 where ***`PreparedEventContent`*** is a placeholder that needs to be substituted with the result of the following preparation process:

1. Take the Event Content string as captured by the regular expression subgroup in Listing 35.

2. Normalize it according to CADP's `bcg_write` normalization rules for string values as specified in (CADP manual authors 2017b).

3. Escape sequences of characters that have a special meaning in regular expressions according to the specification in (CADP manual authors 2017h), i.e. prepend each occurrence of the characters `\`, `[`, `*`, and `.` with a backslash.

4. Escape each occurrence of a single quote character (`'`) by prepending it with a backslash.

Once the placeholder is substituted with a string according to these rules, the overall resulting MCL expression describes a regular expression-based "action predicate". This predicate is satisfied by a transition of the respective given LTS if its string representation matches the regular expression within the two single quotes at the start and the end of the expression.

5.4 User-Perspective Requirements for an Implementation

We describe user-perspective requirements (RU) for an implementation of the theoretical approach described in previous sections. An implementation must fulfill these requirements to enable users to apply our theoretical approach in practice. Our requirements assume that an implementation of an [em]-like meta modeling tool with the data model as described in section 4.1 exists as a foundation for the implementation of our approach. The requirements are as follows:

RU 1. A user must be able to specify sequences of behaviors with our ESDL as described in section 4.4 for Models and ElementOccurrences on the level of the meta modeling tool's Languages and their ElementTypes, respectively.

RU 2. A user must be able to specify the fields of the formulaic expression language's custom types, i.e. of CustomStorageData and of CustomEnablementData.

RU 3. A user must be able to initiate checking the fulfillment of temporal properties by the meta modeling tool's Models in Languages and with ElementTypes for that sequences of behaviors have been defined.

RU 4. Once model checking is completed, the user must be informed about its result, both as a Boolean value representing fulfillment or non-fulfillment of the given property, and as a graph of model elements that are "responsible" for the result as described in section 3.2.

Additionally, an implementation of our approach should have a good usability, i.e. users should be able to easily apply our approach in practice with the implementation. We recognize that this is not a strict or precise requirement. To illustrate our point of view on usability, we outline our usability considerations for our implementation in section 6.5.

6 Implementing the Approach and Integrating it into [em]

In this chapter, we describe our implementation of the approach developed in the previous chapters as a plugin for [em].

In the first section, we give basic information on our plugin's functionality and implementation. In the second section, we introduce the plugin's architecture. In the third section, we give details about user-relevant data persisted by the plugin. In the fourth section, we introduce the main components of our plugin and describe them in detail. In the fifth section, we name aspects of our plugin that should ensure good usability.

6.1 Basic Details on Plugin Implementation

In this section, we give an overview over the functionality offered by our plugin and explain general aspects about its implementation.

Our plugin implementation fulfills the requirements from section 5.4. It allows defining formal temporal properties in our macro-extended version of MCL and storing these properties. It transforms models, their meta models and the assigned behaviors into LNT. It transforms macro-extended temporal properties into plain MCL properties. It launches CADP tools, especially the CADP model checker to perform model checking with the LNT specification and the MCL property. It translates model checking results back into a format that allows highlighting element occurrences that events relevant for proving the fulfillment or nonfulfillment of a property were reported for.

The provided functionality is available from our plugin's easy-to-use user interfaces that are integrated accessibly into the main [em] user interface.

The meta modeling tool [em] is implemented in C#. While plugins for [em] can be implemented in any .NET language, we decided to implement our plugin in C# as well because we were most familiar with this language.

Multiple parts of our implementation rely on the "visitor pattern". We give a brief description of this pattern. An implementation of visitor pattern defines an abstract visitor interface that provides a method signature for each kind of class whose instances may be "visited". Each of such instance may be asked to accept a visitor. When asked to accept such a visitor, the instance calls the visitor's method that corresponds to its class. This pattern allows to implement class-specific external behavior in an object-oriented way. A more detailed description of this pattern is available in (Gamma et al. 1994, pp. 331–344) .

For the architecture behind the user interface of our implementation we made use of the "Model-View-ViewModel" pattern. (Smith 2009)

6.2 Overview of Plugin Architecture, Persistent Data, and Data Flow

In this section, we describe the architecture of our plugin and explain how it integrates into [em] by describing its high-level data flow through its components. A visualization of the architecture and the high-level data flow is given in Figure 11. For the visualization, we extend our notation as introduced in 3.3: We use boxes with sharp corners to represent high-level components. Dotted connections with a black circle as target connector represent that the source element is attached to the target element. Green elements are native [em] components, blue elements are components we developed in our plugin, and orange elements (used in later figures) are CADP components.

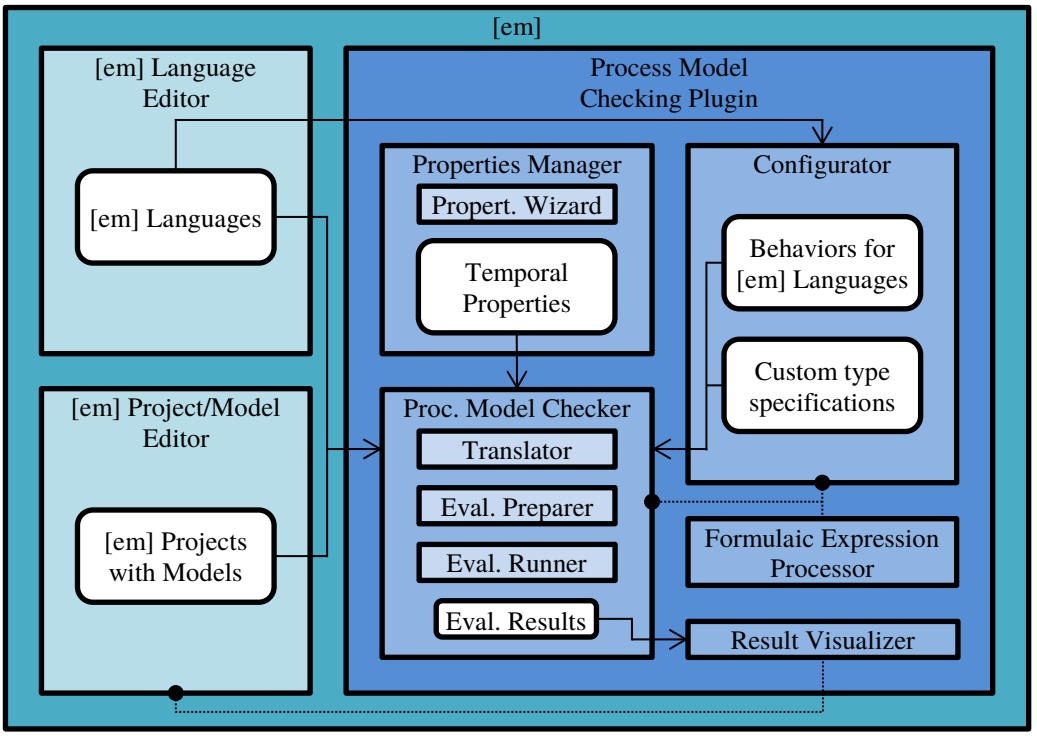

Figure 11 The plugin architecture and its high-level data flow

Our plugin is composed of five main components: the Properties Manager, the Configurator, the Process Model Checker, the Result Visualizer and the Formulaic Expression Processor.

The Properties Manager component provides a user interface for specifying temporal properties and storing their specifications. It also provides the Temporal Property Specification Wizard sub-component (abbreviated with “Propert. Wizard” in Figure 11) that allows a user to generate specifications of selected temporal properties by filling out a questionnaire.

The Configurator component provides a user interface for specifying the two custom types CustomEnablementData and CustomStorageData and for defining behavior sequences for [em] Models and Element Occurrences on the Language level. It loads the [em] Languages to provide information for the user interface that are required during the definition of behavior sequences.

The Process Model Checker (abbreviated with “Proc. Model Checker” in Figure 11) is the core component of our plugin. It provides a user interface for creating and issuing model checking tasks. A model checking task consists of a reference to an [em] Model, a temporal property specification, and additional parameters. When a model checking task is to be performed, the Process Model Checker’s sub-components Process Model Translator (abbreviated with “Translator” in Figure 11), Evaluation Preparer (abbreviated with “Eval. Preparer”), and Evaluation Runner (abbreviated with “Eval. Runner”) perform the required work. They read the referenced [em] Model, its parent Project and all other Models in the Project, their Languages, their assigned behaviors, and the specified custom types. From these pieces of information, they prepare LNT code for the CADP model checker, and start the model checking process for the respective model checking task. Once a model checking task completes, the Process Model Checker passes its Evaluation Results (abbreviated with “Eval. Results” in Figure 11) to the Result Visualizer component. To avoid unnecessary re-computations, the Process Model Checker orders model checking tasks in such a way that the CADP tools can re-use data that was generated for the respective previous task.

The Result Visualizer component receives model checking results from the Process Model Checker to visualize the counterexample or witness information as a tree in a sidebar of [em]’s Model Editor and by highlighting Element Occurrences in the Editor itself. To display the tree and to highlight Element Occurrences, the Result Visualizer attaches itself to [em]’s Model Editor.

The Formulaic Expression Processor processes formulaic expressions to return data that can be used to assist the user in expression specification and to convert formulaic expressions into LNT. The formulaic expression processer is used within the Configurator to provide user assistance, and within the Process Model Checker to provide data required for LNT translation.

6.3 Details on Plugin’s Persistent User Data

In this section, we introduce the types of user-relevant data that our plugin persists. We describe the classes and their structure we created for data that is to be persisted.

Temporal Properties. To represent temporal properties, we created the class `ModelProperty`. This class has three string properties: one for a name for the property, one for its description, and one for the property specification as specified in our macro-extended version of MCL as described in subsection 5.3.1.

Custom Types. To represent the specifications for a Custom Type, we created the classes `CustomDataType` and `CustomDataTypeEntry`. We store names of custom type fields and their data types as string properties in `CustomDataTypeEntry`. We store a collection of `CustomDataTypeEntry` instances and a timestamp of the latest change in `CustomDataType`.

Behaviors. To represent ESDL behaviors, we created a class for each ESDL behavior type. We wanted our software architecture to ensure that all behavior types would be handled by relevant components, such as the user interface or the LNT Generator as described in subsection 6.4.3. As such, we implemented a visitor pattern with a `IBehaviorVisitor` interface that contains one method signature for each behavior type. Code that needs to handle behaviors is consequently forced to implement all methods of this interface. This reduces the risk of forgetting to implement the logic for some of the behavior types.

Behaviors Carriers. To represent an assignment of behavior sequences to models and their elements on the language level, we created the class `BehaviorsCarrier`. Our plugin allows to maintain a set of `BehaviorCarrier`s for each language. In instances of this class, we store the name of the assignment's type as a string value, a modification date, and an ordered collection that represents the sequence of behaviors. To assign a behavior sequence to the model of the respective language, the type name needs to be set to *null*.

Formulaic Expressions. Our behavior objects store formulaic expressions as string properties. These formulaic expressions are only parsed into a better processable representation when necessary, e.g. when editing them in the user interface and when translating them for the formal process specification for the underlying model checker. This design decision allows a user to store a syntactically incorrect formulaic expression. The ability of storing an incorrect formulaic expression can be convenient for cases where a user must interrupt her work on a complex formulaic expression and wants to store the current progress to continue working on it later.

6.4 Details on Plugin's Core Components

In this section, we describe the core components and sub-components of our plugin in detail. For the order of introduction, we roughly follow a workflow that a user might perform when using our plugin and its components. We use our workflow visualization notation as introduced in section 6.2 for visualizing workflows in this section.

In the first subsection, we introduce the Temporal Property Specification Wizard. In the second subsection, we describe how we implemented processing of formulaic expressions in the Formulaic Expression Processor. In the third, fourth and fifth subsection, we introduce the Process Model Translator, the Evaluation Preparer, and the Evaluation Runner, respectively.

6.4.1 Temporal Property Specification Wizard

In (Remenska 2016, chap. 5), a "property assistant tool" called "PASS" was introduced to simplify specification of requirements for event-based systems. "PASS guides users through the elicitation process by asking questions, and providing a set of alternative answers to choose

from, narrowing down the scope of the questions to those relevant in the context of the previously provided answers in each subsequent step." (Remenska 2016, p. 74)

PASS outputs temporal properties in a property specification language that is very similar to MCL. This lead us to the realization that we could easily adapt the concept of PASS to implement a Temporal Property Specification Wizard in our plugin, allowing users of our plugin to easily specify temporal properties through PASS' questionnaire and its temporal property patterns.

Daniala Remenska, the creator of PASS, gave us permission to adopt the PASS questionnaire with its patterns in our work. While the property specification language used for PASS' temporal property patterns is similar to MCL, not all PASS patterns would yield valid MCL properties. As such, we had to adjust some patterns in our implementation so that they yield valid MCL properties. To clearly separate her work from ours, we put our final adaptation into its own library and only referenced to it from our other libraries.

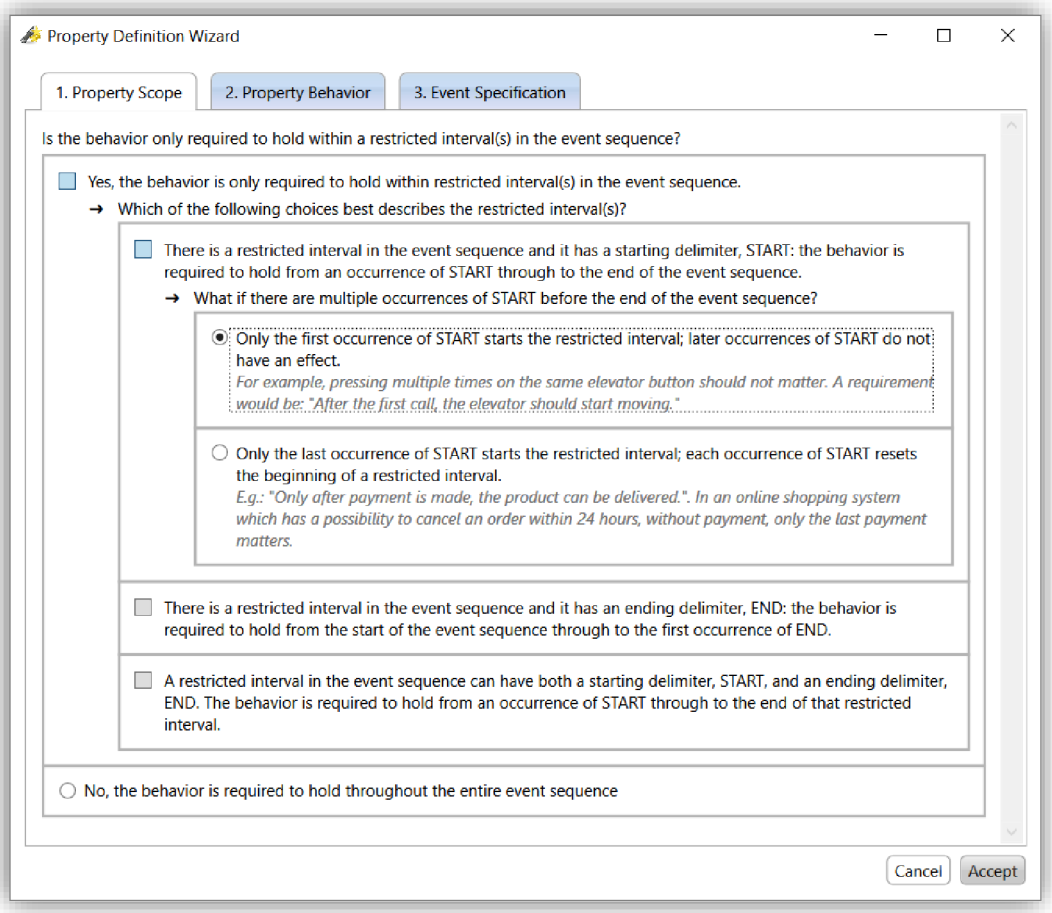

Figure 12 Screenshot of our plugin's Temporal Property Specification Wizard

Our plugin's Temporal Property Specification Wizard provides a user interface that displays the questions in the external library's questionnaire and guides the user through these questions.

A screenshot demonstrating how this user interface is presented to a user is given in Figure 12. Once a user answers all questions and completes the questionnaire, the Wizard uses the adaptation library to translate the answers into a temporal property that corresponds to the user inputs.

6.4.2 Formulaic Expression Processor

Our Formulaic Expression Processor processes expressions in our formulaic expression Language as introduced in section 4.2. This component takes a formulaic expression in our language as its main input. As output, it yields information that further components of our plugin require, especially an abstract syntax tree (AST) corresponding to the input expression.

In this section, we explain why we implemented our lexer/parser on our own instead of leveraging existing implementations, e.g. libraries or parser generators, and describe the detailed processing workflow of the Formulaic Expression Processor.

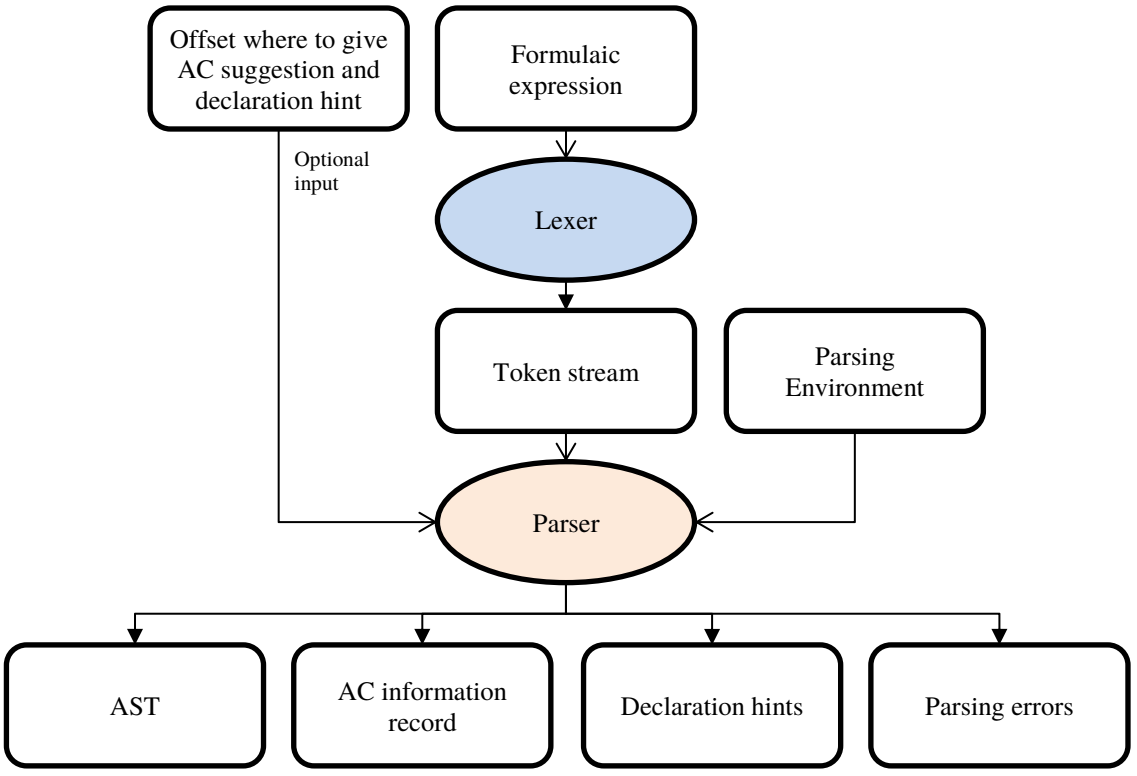

Figure 13 Workflow of the Formulaic Expression Processor

As one of our usability considerations, we wanted our implementation to support the user in specifying formulaic expression with an autocompletion (AC) feature. We give more details on our usability considerations and on the AC feature in section 6.5. With the intention of avoiding implementation work for a lexer/parser combination, we did an online search for solutions that would allow easy creation and integration of AC-supporting lexers/parsers. We also posted a question on the StackExchange community “Software Recommendations”. (Pribnow 2016b)

Neither our online search nor the posted question yielded a suitable solution. We therefore implemented a custom lexer/parser combination on our own.

The resulting lexer/parser combination forms the core of our Formulaic Expression Processor. We describe the component's processing workflow in the following. A visualization of this workflow is given in Figure 13.

Initially, the **lexer** splits a formulaic expression into a sequence of tokens. A token is either a literal (Boolean, double, integer, or string), an opening or closing parenthesis, a full stop, a comma, a whitespace, an opening or closing pointy bracket, a vertical line (also referred to as "pipe symbol"), an identifier, or an unknown token. We implemented the visitor pattern for tokens: In our implementation, each token object can accept a token visitor. When a token is requested to accept a token visitor, it calls a method of the token visitor corresponding to the token's type. This design forced us to handle all token types during further processing steps, thereby lowering the risk of bugs in our implementation.

Next in the workflow, the **parser** takes as input a "parsing environment" and a sequence of tokens from the lexer, and optionally an offset of a token within the formulaic expression for which an autocomplete suggestion and a type hint should be returned. We explain the parser's outputs and the "parsing environment" concept further below.

The parser's state is mainly managed using a stack of parser token visitors. Such a parser token visitor is for example "expect a new Base", "have a valid formulaic expression, so expect a full stop for an Accessor", or "just got a full stop, so expect the Accessor's identifier". The parser iteratively asks the input tokens from left to right to accept the respective topmost token visitor on the stack. When a token visitor visits a token, it modifies the stack so that it reflects a new state.

For example, an "expect a new Base" visitor handles a literal or an identifier to produce a corresponding node in the AST and to replace the topmost visitor on the stack with a "have a valid formulaic expression, so expect a full stop for an Accessor" visitor. An "expect a new Base" visitor would pass handling of any other type of token than a literal and an identifier to the next deeper visitor on the stack. Similar, a "have a valid formulaic expression, so expect a full stop for an Accessor" visitor only handles a full stop token on its own, replacing the topmost visitor again, and falls back to the next deeper visitor on the stack for all other token types.

A visitor may be requested to handle an unexpected token. For example, an expression requiring autocompletion for example might not agree with our language specifications completely and would result in an unexpected token sequence. We wanted our parser to allow for AC and for meaningful errors for parts of the input expression that come after the first unexpected token. We therefore required our parser to be somewhat "forgiving" when encountering unexpected

tokens. As such, we designed our parser as follows. If a visitor visits an unexpected token, it will produce an error message and will replace the topmost visitor on a stack with a “invalid token handling” visitor. Such a visitor silently skips most tokens. When visiting a token that indicates syntactic correctness being restored from that token on, the visitor replaces the topmost visitor on stack again with a suitable “normal” visitor, allowing for normal continuation of parsing.

The parser generates as output: an AC information record, a list of declaration hints, a list of parsing errors, and – if the input formulaic expression is error-free – an AST corresponding to the input expression. We explain these results in the following.

An **AC information record** consists of AC suggestions, the user inputs that are to be autocompleted, and further information required for replacing these user inputs if some suggestion is accepted. An AC suggestion contains information with what the respective part of the user inputs should be replaced. It also contains additional information that further explains the AC suggestion to a user, like usage information for the suggestion and information on its result type. These additional pieces of information can be displayed together with the suggestion to a user as exemplarily shown in Figure 14 .

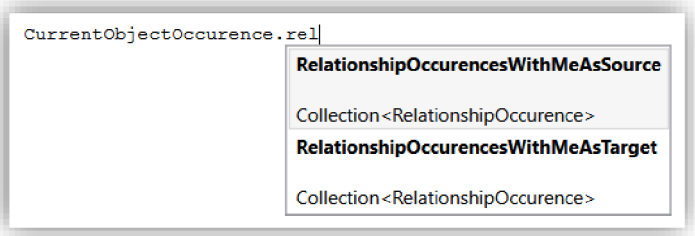

Figure 14 Screenshot showing our implementation’s autocomplete suggestion feature

A **declaration hint** allows a user to quickly find out how a part of a formulaic expression can be used and what type it would result in. When a user hovers with the mouse over an identifier, a constant or an accessor, she will be shown a tooltip window displaying the result type or the whole declaration of the respective part of the formulaic expression. This feature was inspired by code editors of modern integrated development environments like Visual Studio that support such a feature for their main supported computer languages.

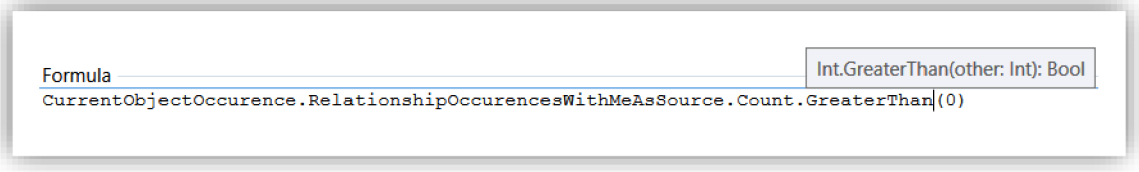

Figure 15 Screenshot showing how a declaration hint is displayed by our plugin

A screenshot demonstrating how a declaration hint is displayed in our implementation is given in Figure 15. For function accessors, we display the base type name, accessor identifier, function parameter information, and return type information. For lambda accessors, we display the base type name, accessor identifier, lambda parameter information, expected lambda sub-formula result type, and information on the return type of the overall lambda accessor application. For property accessors, we display the base type name, the accessor identifier and the result type name. For base identifiers, we display the identifier name and their type.

A **parsing error** allows to inform the user on errors that occurred during parsing, typically indicating a malformed expression. To allow users to quickly find reasons for parsing errors and fix them, we wanted to make it possible to automatically select a part of the expression in a user interface for editing formulaic expressions that caused some parsing error to occur. In our implementation, we therefore also keep track of the offset and length of the respective input expression's part causing the parsing error.

To provide declaration hints and meaningful AC suggestions, knowledge of a formulaic expression's result type is required. We found it easiest to implement the generation of these pieces of information in the parser. However, introducing type information into our parser would have violated the separation of concerns design principle. So instead of directly integrating them into our parser, we defined the abstract class "**parsing environment**" whose implementation's instances are used by the parser to retrieve information required for generating an AC information record and for declaration hints. We realized that we can save further implementation effort by delegating also the generation of AST nodes to the parsing environment. This way, we avoided implementing another post-processing step to derive result type information from the AST for further processing. In the final implementation, the parser itself consequently does not generate any nodes for an output AST. Instead, it delegates this task to the parsing environment, thereby yielding an AST with type-rich nodes.

Like in our implementation of behaviors of tokens, we implemented the visitor pattern for data types and for their members that can be referred to using accessors in formulaic expressions. We wanted to ensure handling all types and all their members to reduce the risk of bugs in our implementation, especially during development of the LNT Generator as described in subsection 6.4.3 that implements the process outlined in section 5.2.

To manage types for formulaic expressions in our language, we defined a multi-level type hierarchy that has a set of the primitive types as its root. On higher levels of this hierarchy we implemented the [em] types and the runtime-relevant types. The multi-level type hierarchy design allows extending the type system without having to modify lower hierarchy levels if a newer version of our formulaic expression language requires new types.

6.4.3 Process Model Translator

The Process Model Translator takes as input a set of [em] Projects with their Models and their Languages, user-defined custom type specifications, and user-defined behaviors assigned to ElementOccurrences and Models on the meta model level. From these inputs, it generates a formal process specification in a language that the underlying model checker can work with.

Our theoretical approach does not limit the number of Projects that could be processed in one model checking task. In practice however, working with many Projects with many Models and many Languages causes long processing times. Lacking support for cross-project links on the level of Elements or Models, the [em] data model suggests that everything in a project is to be considered conceptually separate from other projects. In our final implementation we therefore allow the Process Model Translator to only process a single Project per model checking task. Consequently, it generates one set of formal process specifications per Project as output.

We describe the detailed translation workflow of the Process Model Translator in the following. A visualization of this workflow is given in Figure 16.

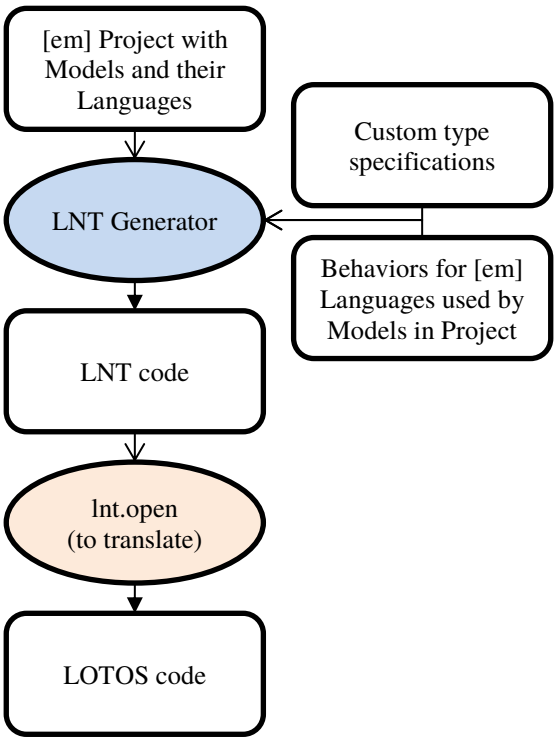

Figure 16 The Process Model Translator's workflow

As first step of translation, the three inputs get passed to our LNT Generator component. The LNT Generator is an implementation of the process outlined in section 5.2. It yields a translation of our input process models as files containing formal processes specifications in LNT. Since [em] supports the Unicode character set, but CADP's LNT implementation only supports the

ASCII character set, our implementation must replace all non-ASCII characters in fields of the [em] data model's classes with a placeholder. We use a question mark as our placeholder.

As of writing this thesis, a formal process specified in LNT needs to be translated to LOTOS before the CADP toolkit can perform model checking on the formal process. In the next step, our translator subsequently translates these LNT-specified formal processes into files that contain formal processes specified in LOTOS by executing CADP's `lnt.open` tool, using its pre-processing and translation operations. This tool internally calls CADP's `lpp` and `lnt2lotos` tools for the actual pre-processing and translation. The resulting LOTOS files are the main input of the Evaluation Preparer's workflow.

6.4.4 Evaluation Preparer

Given an [em] Model that should initially be enabled in a model checking task, the Evaluation Preparer takes as input the [em]-internal ID of this Model, and the LOTOS files for the Model's Project as generated by the Process Model Translator. The Evaluation Preparer generates an Evaluator Executable file that can be called to perform the actual model checking operations for the given Model. We describe the detailed executable generation workflows in the following. A visualization of these workflows is given in Figure 17.

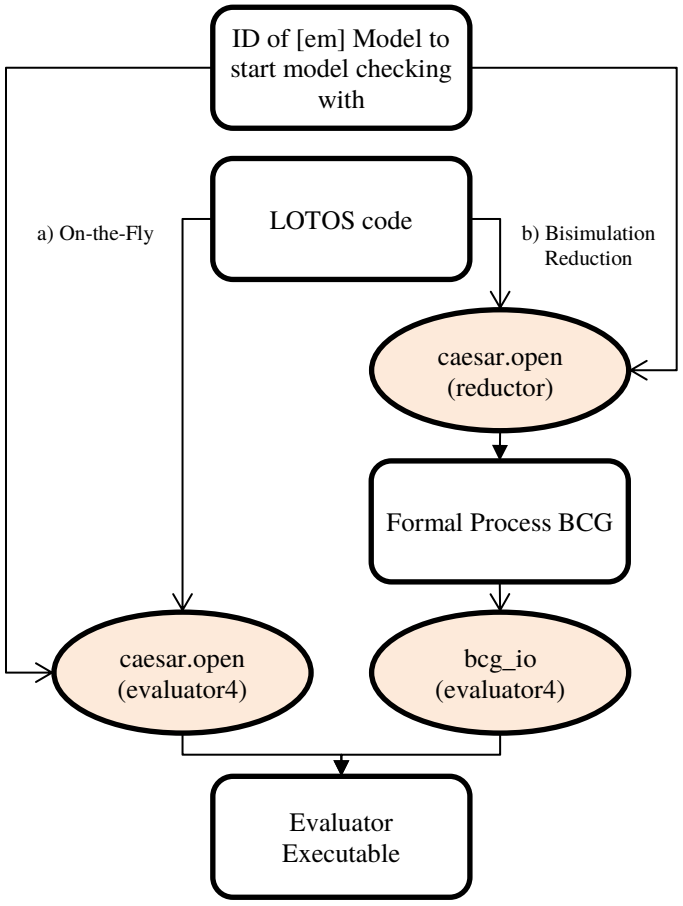

Figure 17 The Evaluation Preparer's workflows

The Evaluation Preparer can be run in two different modes: the On-the-Fly mode and the Bisimulation Reduction mode.

In the On-the-Fly mode, the LOTOS files are processed directly, i.e. only so much of the LTS of the formal processes is computed during evaluation as required to prove or disprove the fulfillment of the property. This can save a significant amount of otherwise required pre-processing time and should lead to getting results fast, especially when asked to disprove that some chain of events is never to occur. On the other hand, the generated counterexample or witness information can be hard to read for a user because of its complexity in models with a lot of branching.

In the Bisimulation Reduction mode, the LTS are reduced with respect to CADP's "safety" equivalence as specified in (CADP manual authors 2017d). To perform the reduction, the whole LTS of the formal process must be generated. This can increase the pre-processing time significantly but might make the generated counterexample or witness information more readable in models with a lot of branching. For some combinations of Model and temporal property, the evaluation may run faster on a pre-reduced LTS.

The workflows executed by the Evaluation Preparer depend on the given mode.

When using the On-the-Fly mode (indicated with a small letter 'a' in Figure 17), the Evaluation Preparer executes CADP's `caesar.open` tool with the input LOTOS files, with an entry point specification derived from the given input Model ID, and with a reference to the `evaluator4` OPEN/CAESAR tool (CADP manual authors 2017e) that provides CADP's model checking functionality. `caesar.open` internally calls CADP's `caesar.adt` and `caesar` tools as well as the configured C compiler, thereby generating the Evaluator Executable.

When using the Bisimulation Reduction mode (indicated with a small letter 'b' in Figure 17), the Evaluation Preparer executes `caesar.open` tool mode with the input LOTOS file, with an entry point specification derived from the given input Model ID, and with the reference to the `reductor` OPEN/CAESAR tool (CADP manual authors 2017g) that provides CADP's bisimulation reduction functionality. Again, `caesar.open` internally calls CADP's `caesar.adt` and `caesar` tools as well as the configured C compiler, this time generating an executable to generate a reduced version of the input process. The Evaluation Preparer calls the generated executable with arguments to perform a total safety reduction, yielding a file with a respectively reduced LTS in CADP's Binary Coded Graph (BCG) format (CADP manual authors 2017c). Next, the Evaluation Preparer calls CADP's `bcg_open` tool with the path to the BCG file containing the reduced LTS and with a reference to the `evaluator4` OPEN/CAESAR tool. `bcg_open` then generates the Evaluator Executable.

The executable resulting from both workflows becomes the main component that is used by the Evaluation Runner.

6.4.5 Evaluation Runner

The Evaluation Runner takes as input a property in our MCL extension as described in subsection 5.3.1 and an evaluation executable that was generated by the Evaluation Preparer. It yields a Boolean result reflecting the fulfillment of the property by the respective Model, and counterexample or witness information as provided by CADP's `evaluator4` in a representation that can easily be used by our plugin's Result Visualizer. We describe the detailed evaluation and result interpretation workflows in the following. A visualization of these workflows is given in Figure 18.

As a first step of the Evaluator Runner's workflow, the Macro Expander expands macros as described in subsection 5.3.1 to derive a corresponding plain MCL temporal property from the input property that is specified in our extended version of MCL. The Macro Expander is an implementation of the process described in subsection 5.3.2. Since CADP supports only the ASCII character set, all non-ASCII characters of the input property specification will be replaced with a question mark as described in subsection 6.4.3.

After converting the temporal property from our MCL extension into plain MCL, the Evaluation Runner calls the Evaluator Executable generated by the Evaluation Preparer and provide the plain MCL property as input. The Boolean value that is first result from the call becomes the first output of the Evaluation Runner. The Diagnostics BCG file that is the second result from the call contains information from which counterexample or witness information can be derived.

The binary BCG file format does not have a public documentation. Also, there are no .NET-compatible libraries known to us that allow reading BCG files. We could therefore not easily implement a mechanism to read the Diagnostics BCG files. Instead, the Evaluation Runner converts the Diagnostics BCG file into the AUT format (CADP manual authors 2017a) by executing CADP's `bcg_io` tool. AUT being a publicly documented text-based format, it was much easier for us to implement a reading mechanism for this file format.

The AUT Reader uses our AUT reading mechanism to read the counterexample or witness graph in a representation that can be easily be processed by the Result Visualizer.

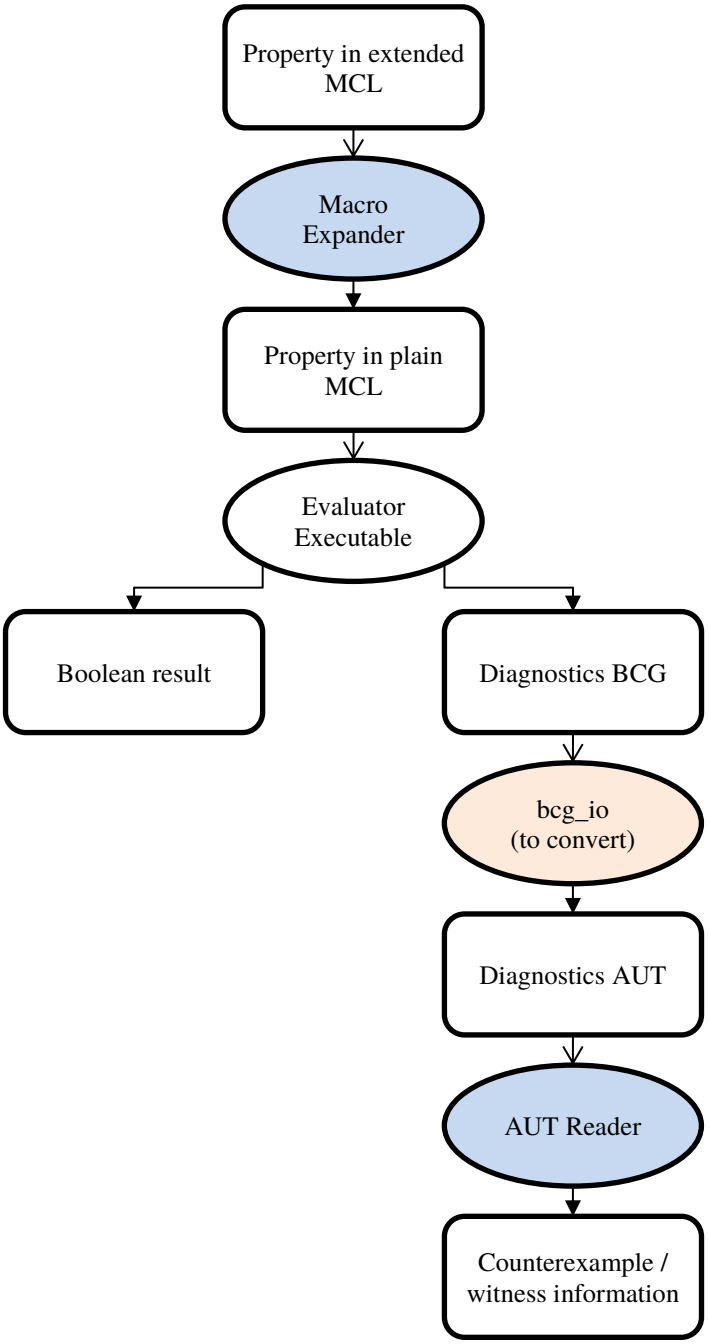

Figure 18 The Evaluation Runner's workflow

6.5 Usability Considerations

As established in section 5.4, an implementation of our approach should have a good usability. In this section, we introduce our usability considerations (UC). We assume that the application of our approach becomes more suitable and easier for users if the used implementation follows these considerations. As such, a user-targeted implementation should also follow our usability considerations.

We introduce the considerations in the following, each with a short explanation on how our implementation follows the respective consideration.

UC 1. A user should be supported by the implementation to specify temporal properties easily.

To simplify temporal property specification, our implementation includes the Temporal Property Specification Wizard described in subsection 6.4.1 that guides a user through a questionnaire and that finally generates a temporal property based on the user inputs.

For supporting the user in manual specification of temporal properties, our implementation contains an easily accessible "cheat sheet" as shown in Figure 19. This cheat sheet lists important constructs of the used temporal property specification language MCL as well as an exemplary temporal property that makes usage of several of these constructs.

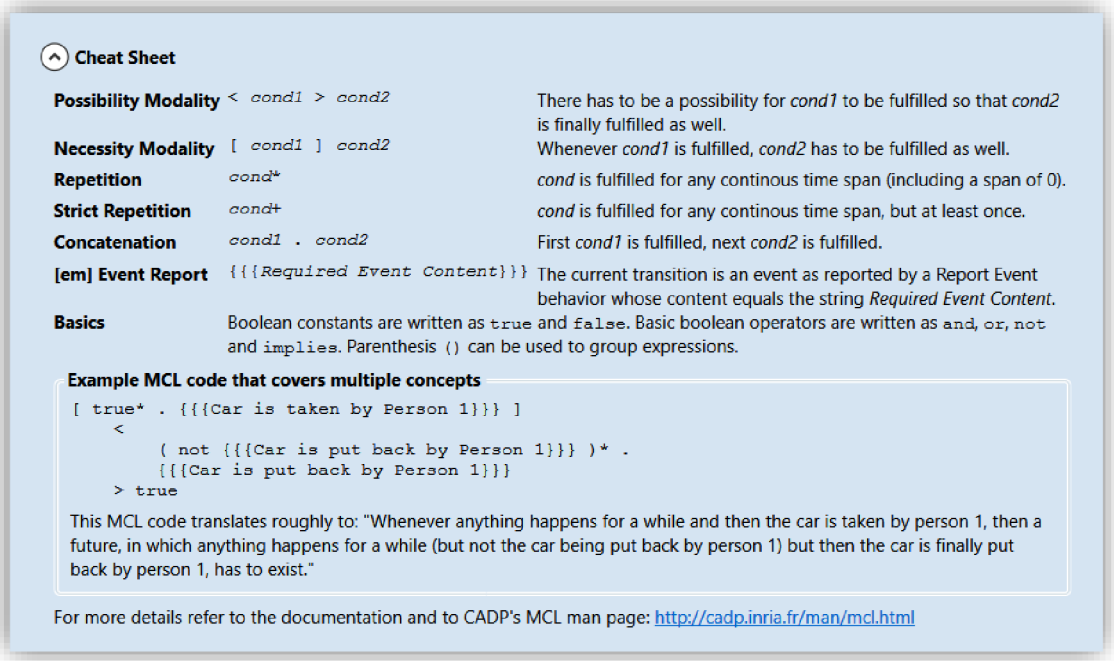

Figure 19 Screenshot showing our temporal property specification cheat sheet

UC 2. A user should be able to specify sequences of behaviors using an easily usable and supportive user interface.

In our implementation, we visualize sequences of behaviors as lists of graphical blocks where the blocks represent behavior instances. A user can re-order these sequences by dragging the blocks with the mouse and dropping them at some different position in the list. Specifying sequences of behaviors in our implementation is like working with a graphical programming language. It can be compared to development of macros in Microsoft Access Web Apps (Microsoft Corporation 2017a), or to building procedures by snapping together graphical blocks in the educational programming environment "Scratch" (Resnick et al. 2003).

When a user specifies a formulaic expression, our implementation supports her with a cheat sheet like the one presented above for temporal properties, and with autocomplete suggestions. We present an example for the autocomplete feature in action. Given a formulaic expression `CurrentObjectOccurrence` that evaluates to an ObjectOccurrence, assume that the user wants to extend this expression so that it evaluates to the RelationshipOccurrences that have the original ObjectOccurrence as source. Then the user would have to append a dot to the end of the original expression, followed by `RelationshipOccurrencesWithMeAsSource`.

Our implementation allows the user to avoid typing out this whole string. Instead, if a user begins typing the first few characters of this long string, a list with suggestions for possible completions of the typed characters will appear as shown in Figure 14 (on page 60). We implemented a special ordering algorithm for ordering suggestions that was inspired from modern integrated development environments. If a user clicks on one of the suggested completions, the typed characters will be replaced with the suggestion. We assume that the autocomplete suggestion feature increases efficiency in specifying formulaic expressions.

UC 3. For automatic procedures with long processing times that need to be executed multiple times with different inputs, a user should be able to define tasks for the required procedures and inputs, and to initiate the collective automatic execution of these tasks.

Because model checking is a computation-intensive activity, it can take a large amount of time to check if a process model fulfills a temporal property. We assume that users often want to collectively check the fulfillment of multiple properties in multiple models. To avoid the user initiating each check one after another manually, our implementation allows to define “model checking tasks” and to initiate the collective execution of these tasks. A model checking task describes the process of checking the fulfillment of a defined temporal property in a defined model.

In our implementation, a user can select multiple models and multiple temporal properties at once and generate “model checking tasks” for all combinations of selected models and selected temporal properties. When initiating the execution of these tasks, the user must not interact with her computer anymore to complete the model checking tasks. She can then for example let the computer perform the required computations overnight.

7 Demonstration

In this chapter, we demonstrate how our approach can be put into practice using our [em] plugin. We present three exemplary artificial case studies by introducing sample languages and models in these languages. To make our case studies more tangible, we placed them into the context of nuclear reactors as a demonstrative setting. Since we are no experts for nuclear reactors, our case studies do not have the intention to represent real-world scenarios.

In the first case study, we demonstrate how execution semantics for a simple business process modeling language can formally be specified with our approach and how our implementation makes witness and counterexample information available to a user. In the second case study, we develop a setting that might resemble a real-world scenario with a more complex model in a more complex modeling language, and use it to demonstrate how our approach and our Temporal Property Specification Wizard can help in identifying problems in process models. In the third case study, we illustrate the cross-model and cross-language analysis capabilities of our language.

7.1 Case Study 1

In our first case study, we introduce a simple business process modeling language with simple execution semantics. We show how behaviors can be assigned to this language and its element with our approach for describing the execution semantics to enable model checking of business process models in this language. We show how the instances of the behavior types Report Event, If/Then/Else, and Enable Element Occurrence may be used in practice. We demonstrate the usage of simple formulas in the behaviors. We finally describe that our approach enables model checking and demonstrates how counterexample information are represented by our implementation.

In the first subsection, we introduce the Simple Linear Process Language (SLPL) that forms the basis of the remaining case study. In the second subsection, we show how SLPL's execution semantics can be translated into sequences of behaviors assigned to SLPL models and their element occurrences on the meta-level. In the third subsection, we describe how model checking can be performed with our approach and show how our implementation visualizes witness information in an interactive fashion.

7.1.1 Introduction into Simple Linear Process Language (SLPL)

We introduce the *Simple Linear Process Language* (SLPL), a process modeling language created for our case study. SLPL has two element types: the object type *Node* and the relationship type *Node Connection*, allowing to connect one *Node* to another. We visualize

Nodes as cyan boxes and *Node Connections* as arrows between two *Nodes*. We give an example business process model in SLPL in Figure 20.

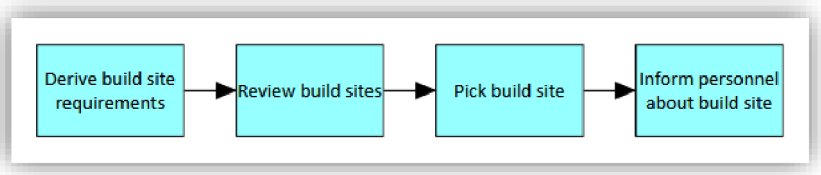

Figure 20 An exemplary business process for the build preparation of a nuclear reactor

If we want to put our approach into practice on SLPL business models, we must specify the SLPL semantics first. For brevity, we gave SLPL very simple semantics: A formal process is derived from a SLPL model from the events that are reported when enabling the model. If a SLPL model is enabled, it must enable the first found *Node* that does not have any incoming *Node Connections*. If a *Node* is enabled, it must a) report an event corresponding to its label and b) enable the first found outgoing *Node Connection*. If a *Node Connection* is enabled, it must enable the target *Node*.

7.1.2 Specification of Behavior Sequences for SLPL

We can now use the execution semantics specified in natural language in the last subsection to formalize them as sequences of behaviors as described in section 4.4.

We give the specification of the sequence of behaviors for a SLPL *Node* in Table 7. This sequence consists of three behaviors: a Report Event behavior and an If/Then/Else behavior that contains an Enable Element Occurrence behavior.

Our semantics require that an event should be reported when enabling a *Node*. This requirement is fulfilled through the Report Event behavior. To capture what element the event was reported by, we used a formulaic expression yielding the current object occurrence as the behavior's "Element Occurrence" argument. To define the content of the public event, we used formulaic expression yielding the label of the *Node* as the behavior's "Event Content" argument.

After reporting the event, our semantics require the first found outgoing *Node Connection* relationship to be enabled. To check if there is any outgoing relationship, we use the If/Then/Else behavior with a formulaic expression as its "Condition" that counts the number of outgoing relationships and checks if this number is greater than 0.

If this check results in $false$, nothing will be done. If the check results in $true$, the "Enable Element Occurrence" behavior will be triggered. We use a formulaic expression yielding the head of the collection of outgoing relationships (i.e. the first outgoing relationship) as the behavior's "Element Occurrence" argument. To put the enablement of the ElementOccurrence

into the scheduler's enablement queue, we set its "Perform now instead of scheduling it" to $false$. This has a computational advantage: If the process model contains a loop and the same ElementOccurrence is to be enabled again at a later point, the model checker can use the scheduler's enablement queue to identify that it already encountered the situation of having to enable the respective ElementOccurrence. In such a case, the model checker may save computational effort because it does not need to re-compute the already computed part of the LTS that results from enabling the respective ElementOccurrence.

Neither does the semantics of our language need to operate with multiple runtime instances nor with stored data. We can therefore make all enablements in our semantics use the initial runtime instance and the initial enablement data. To accomplish this, we use the formulaic expressions as given in Table 7 for all "Runtime Instance" and "Data to pass on" arguments in this example.

Report Event	
Element Occurrence	`CurrentObjectOccurrence`
Event Content	`CurrentObjectOccurrence.Object.Caption`
If/Else/Then	
Condition	`CurrentObjectOccurrence.RelationshipOccurrencesWithMeAsSource.Count.GreaterThan(0)`
Then Behaviors	**Enable Element Occurrence**
Runtime Instance	`CurrentRuntimeInstance`
Element Occurrence	`CurrentObjectOccurrence.RelationshipOccurrencesWithMeAsSource.Head`
Data to pass on	`EnablementData`
Perform now instead of scheduling it	$false$
Else Behaviors	(empty sequence)

Table 7 Behavior sequence for occurrences of SLPL *Nodes*

We give the specification of the sequence of behaviors for a SLPL *Node* in Table 8. This sequence consists of only one behavior: an Enable Element Occurrence behavior. When enabling a *Node Connection*, our semantics require the target *Node* to be enabled. This target

element description translates directly into the formulaic expression used as the Element Occurrence argument.

Enable Element Occurrence

Runtime Instance	`CurrentRuntimeInstance`
Element Occurrence	`CurrentRelationshipOccurrence.TargetElementOccurrence`
Data to pass on	`EnablementData`
Perform now instead of scheduling it	*true*

Table 8 Behavior sequence for occurrences of SLPL *Node Connections*

To complete the behavior specifications for SLPL, we give the specification of the sequence of behaviors for a SLPL model in Table 9. This sequence consists of two behaviors: an If/Then/Else behavior that contains an Enable Element Occurrence behavior.

If/Else/Then

Condition	`CurrentModel`
	`.ElementOccurrences`
	`.Any<eo |`
	`  eo.IsObjectOccurrence`
	`  .And(eo.RelationshipOccurrencesWithMeAsTarget.Count.Equals(0))`
	`>`

Then Behaviors	**Enable Element Occurrence** Runtime Instance: `CurrentRuntimeInstance` Element Occurrence: `CurrentModel.ElementOccurrences.Where<eo | eo.IsObjectOccurrence .And(eo.RelationshipOccurrencesWithMeAsTarget.Count.Equals(0))>.Head` Data to pass on: `EnablementData` Perform now instead of scheduling it: *false*
Else Behaviors	(empty sequence)

Table 9 Behavior sequence for SLPL models

When an SLPL model is enabled, our semantics require the first found *Node* to be enabled that does not have any incoming *Node Connections*. The If/Then/Else behavior checks if such a behavior exists in the model, by using a formulaic expression as the "Condition" argument that operates on a collection of all element occurrences in the model to determine if it contains at least one element occurrence that is a ObjectOccurrence and has no incoming relationships. If such an element occurrence does not exist, nothing will be done. If it does, the If/Then/Else's Then Behaviors sequence will be triggered, i.e. the Enable Element Occurrence.

We use a formulaic expression for the Enable Element Occurrence behavior's "Element Occurrence" argument that operates on the same collection as in the last paragraph. It filters the collection, yielding a collection that contains only ObjectOccurrences without incoming relationship. It then yields the head (i.e. the first element) of the collection.

7.1.3 Model Checking with SLPL Models

With SLPL's semantics defined, we can now demonstrate that model checking with models in SLPL becomes possible. For our demonstration, we use the simple exemplary property "there must be a way that the process runs into an infinite loop" and check its fulfillment by the simple exemplary models as given in Figure 20 (on page 70) and Figure 21.

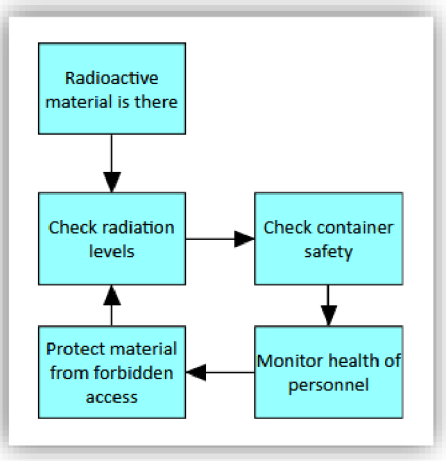

Figure 21 Exemplary SLPL model containing a loop

From looking at the two models, it is obvious that the property is not fulfilled by the first model, but by the second one.

We demonstrate that our approach gives us the correct answers for the two models. We also demonstrate our implementation's visualization of counterexample and witness information. For the demonstration, we need our temporal property in MCL. We give the translation of our exemplary temporal property from natural language into MCL in Listing 38.

```
< true > @
```

Listing 38 Temporal property in MCL that describes the existence of a cycle in the LTS

Our implementation gives us the expected result for each model, i.e. “not fulfilled” for the first one and “fulfilled” for the second one. Additionally, it gives witness and counterexamples information in a stack-like representation. We give our implementation’s representation in Figure 22. The box on the left shows counterexample information for the first model. The box on the right shows witness information for the second model.

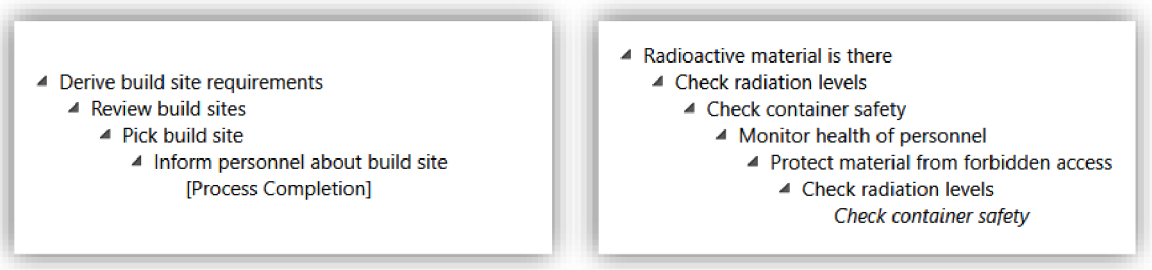

Figure 22 Two exemplary counterexample and witness event chains

The stacks show the reported public events that prove or disprove the fulfillment of the property. If an event transitions to a state that was explored before, its text is written in italics.

Our implementation’s visualization has an interactive component: Double-clicking on an event with italic text changes the selection to the event that led to exploring the known state the first time. Double-clicking on any other event that was reported by an element occurrence opens its model. It then highlights the reporting element occurrence by surrounding it with a colored border.

The border colors depend on the chain of events that lead to the selected event. The selected event’s element occurrence is always highlighted in light green. The chain’s first reported event’s element occurrence is highlighted in blue. All events between these two events are highlighted in dark green in our example. In section 7.3, we demonstrate that “jumps” between models within a chain of events are visualized by using further colors.

In Figure 23 we exemplarily show how our implementation highlights element occurrences when double-clicking on the event “Check container safety” (on the left) and on the event “Monitor health of personnel” (on the right).

We can infer from the witness information for the second model: The second time the event “Check radiation levels” is reported, the LTS transitions via the event “Check container safety” to a previously explored state. This proves the existence of a loop. Using the interactive visualization, we can follow the chain of element occurrences reporting the events that make up the loop. We found double-clicking on events in counterexample and witness event chains

to be helpful for determining how the events and their reporting element occurrences go together.

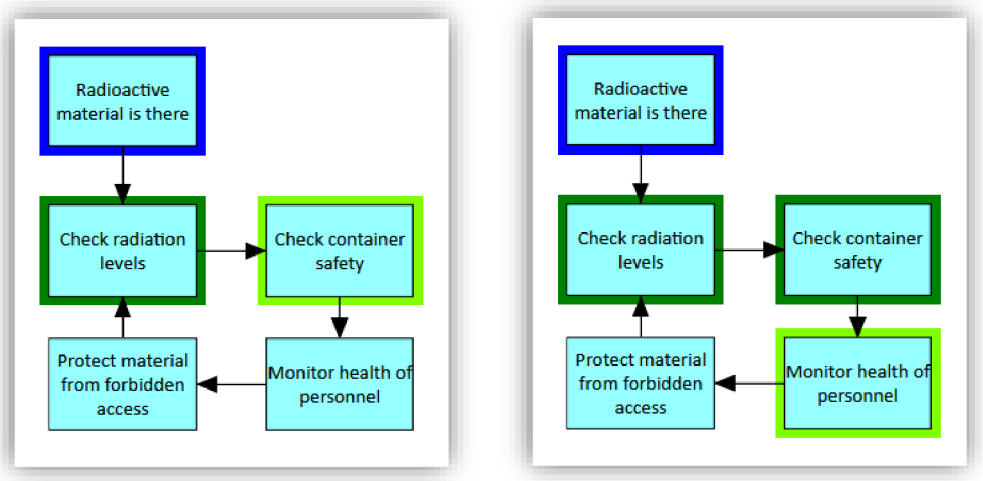

Figure 23 Exemplary highlighting of element occurrences in a witness event chain

7.2 Case Study 2

In our second case study, we introduce a more complex, EPC-inspired exemplary business process modeling language. We show how execution semantics may be specified that reports events corresponding to more than one Element. We demonstrate the usage of more complex formulaic expressions, for example to check the type of an Element. We demonstrate how data storing and loading can be used to implement a merging join of two process branches that run parallel to each other. Our presented execution semantics demonstrates how the behaviors “For one item in collection”, “Load data”, and “Store data” may be used. We further demonstrate how our Temporal Property Specification Wizard can be used to easily generate complex temporal property specifications. Finally, we demonstrate how our visualization of counterexample information may help in better understanding a model checking result.

In the first subsection, we introduce our more complex exemplary business process modeling language and describe how its semantics can be formally specified with our ESDL. In the second subsection, we show how our Temporal Property Specification Wizard helps in specifying a temporal property, and demonstrate how results from our approach may help in finding the sources or reasons for a violation of a property.

7.2.1 Basic Highly Simplified EPC without Interfaces (HSPEC)

We introduce the process modelling language *Highly Simplified EPC without Interfaces* (HSEPC). HSEPC is inspired by EPC (Keller et al. 1992) and uses some of its elements.

HSEPC has nine element types. Its six object types are *Event* (visualized as a hexagon), *Function* (visualized as a box with rounded corners), *Business Object* (visualized as a cut piece of paper), *Competent Body* (visualized as a box with a double left border), *AND* (visualized as a circle containing the symbol ∧), and *XOR* (visualized as a circle containing the symbol ×). Its three relationship types are *Flow* (visualized as an arrow), *Business Object to Function*, and *Competent Body to Function* (both visualized as a line). We give an exemplary HSEPC model with these elements in Figure 24.

The semantics of HSEPC correspond to the informal semantics in (Keller et al. 1992). To allow for model checking, we want occurrences of *Functions* and *Events* on a process flow to report corresponding events like in SLPL. As an extension, we want to include the name of the element's type in our events. For *Functions*, we additionally want to include an associated *Business Object* and an associated *Competent Body* in our events if such objects exist.

For brevity, we do not give the detailed behavior sequences but only explain the most important aspects to them.

The behavior sequence for a *Flow* is the same as for a *Node Connection* in SLPL.

The behavior sequence for an *Event* is very similar to the one for a *Node* in SLPL with one difference: Instead of reporting only the caption of the element, the caption is prepended with the string `Event:` to indicate the reporting element occurrences' type.

```
CurrentObjectOccurrence
      .RelationshipOccurrencesWithMeAsTarget.Any<ro |
            ro.Relationship.Type.Equals
                  (CurrentLanguageElementTypes.RelationshipTypes.Competent_Body_to_Function)
      >
```

Listing 39 Formula to check for an incoming relationship of a specific type

The behavior sequence for a *Function* is similar to the one for an *Event*. To include a potentially associated *Business Object* or *Competent Body* in the reported event, the behavior sequence needs If/Then/Else behavior that checks if such associated objects exist. A condition to check for the existence of an associated *Business Object* is given in Listing 39.

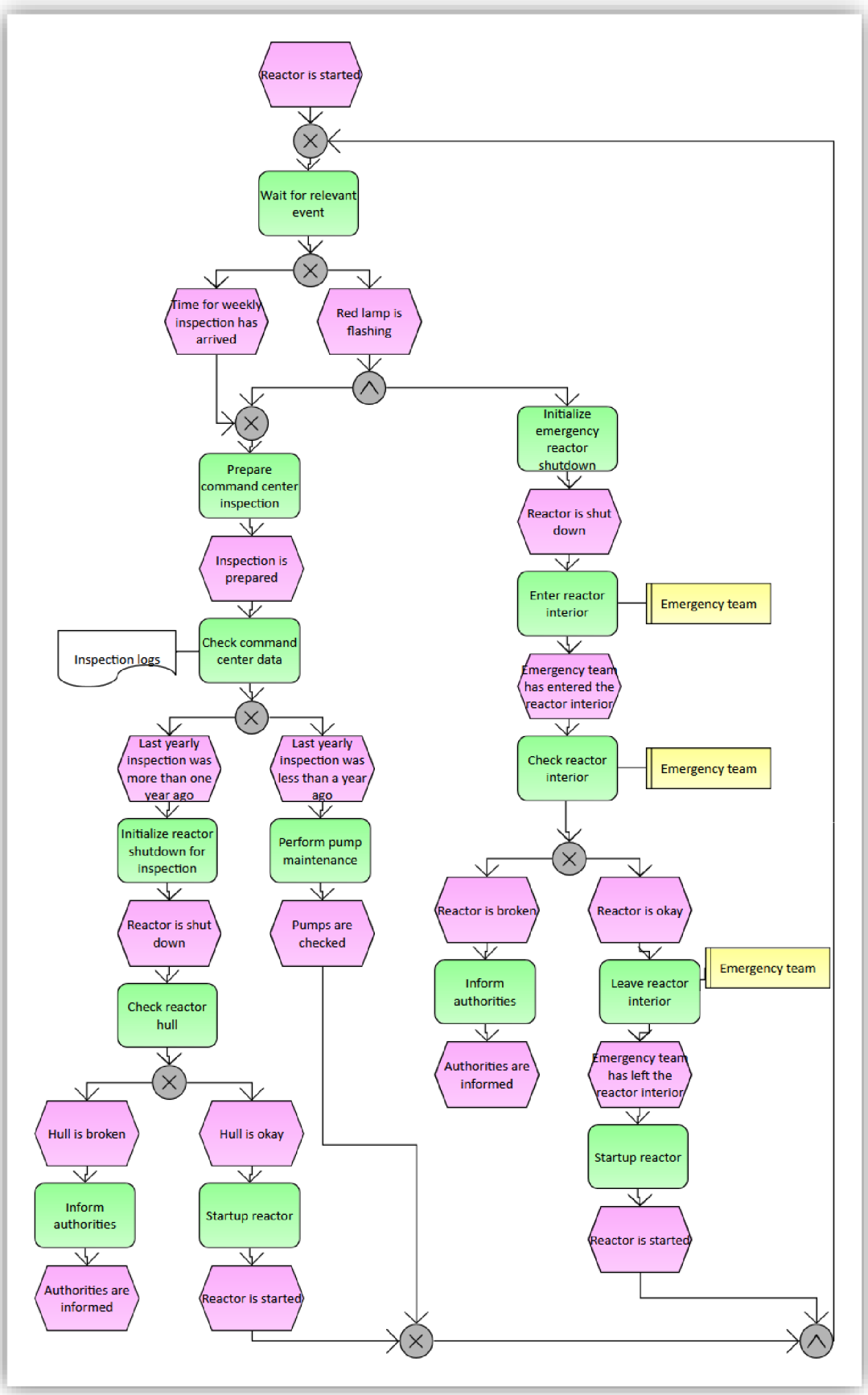

Figure 24 An exemplary HSEPC model

If an associated *Competent Body* exists, the Event Content reported by the *Function* needs to consist of 1) a constant string corresponding to the type name, 2) the caption of the *Function*, 3) the caption of the associated *Competent Body*. We can use string concatenation to combine these pieces of information. In Listing 40, we give a possible Event Content argument for *Functions* that have an associated *Competent Body* but no associated *Business Object*. If a *Function* has both an associated *Business Object* and a *Competent Body*, the formulaic expression must be extended with additional concatenation accessors, respectively.

```
"Function: "
.ConcatedWith(CurrentObjectOccurrence.Object.Caption)
.ConcatedWith(" performed by ")
.ConcatedWith(CurrentObjectOccurrence
      .RelationshipOccurrencesWithMeAsTarget
      .Where<ro |
            ro.Relationship.Type.Equals
                  (CurrentLanguageElementTypes.RelationshipTypes.Competent_Body_to_Function)
            >
      .Head
      .SourceElementOccurrence
      .AsObjectOccurrence
      .Object
      .Caption)
```

Listing 40 Event content formula for HSEPC *Functions* with associated *Competent Body*

The behavior sequence for an *XOR* is given in Table 10.

For one item in collection

Item Variable Name	follower
Collection	`CurrentObjectOccurrence.RelationshipOccurrencesWithMeAsSource`
Child Behaviors	**Enable Element Occurrence**

Runtime Instance	`CurrentRuntimeInstance`
Element Occurrence	`follower`
Data to pass on	`EnablementData`
Perform now instead of scheduling it	*false*

Table 10 Behavior sequence for occurrences of HSEPC *XOR*

In HSEPC, an *AND* keeps track of the number of times it was enabled. If the number of its enablements equals the number of incoming relationships, it will reset the enablement counter and will enable the outgoing relationships in parallel. To implement this concept, our CustomStorageData is defined to contain an integer field *NumberOfRegisteredTriggerings*. The loading and storing part of the behavior sequence for an *AND* is given in Table 11.

Load data

Runtime Instance	`CurrentRuntimeInstance`
Element Occurrence	`CurrentObjectOccurrence`
Variable Name	data
Child Behaviors	**If/Then/Else**

If/Then/Else

Condition	`data.NumberOfRegisteredTriggerings`
	`.Plus(1)`
	`.LessThan(`
	`  CurrentObjectOccurrence`
	`  .RelationshipOccurrencesWithMeAsTarget`
	`  .Count`
	`)`

Then Behaviors	**Store data**
Else Behaviors	**Store data**

Store data (Then Behaviors)

Runtime Instance	`CurrentRuntimeInstance`
Element Occurrence	`CurrentObjectOccurrence`

Data to be stored	`Data`
	`.WithChanged_NumberOfRegisteredTriggerings(`
	`  Data`
	`  .NumberOfRegisteredTriggerings`
	`  .Plus(1)`
	`)`

Store data (Else Behaviors)

Runtime Instance	`CurrentRuntimeInstance`
Element Occurrence	`CurrentObjectOccurrence`
Data to be stored	`Data` `.WithChanged_NumberOfRegisteredTriggerings(0)`

[Behaviors to enable outgoing relationships in parallel]

Table 11 Partial behavior sequence for occurrences of HSEPC *AND*

The element types *Business Object*, *Competent Body*, *Business Object to Function*, and *Competent Body to Function* themselves do not have behavior sequences assigned to them; the behavior sequences of the other element types take them into account.

The behavior sequence for enabling a HSPEC model must enable all *Event* occurrences without incoming relationships in parallel.

7.2.2 Model Checking with HSEPC

The exemplary process model in Figure 24 has a complexity that a real-life process model might also have. We use it to demonstrate our approach working with models that are close to practice. Assume we want to make sure in our process design that the emergency team does not enter the reactor while it is running. We use the property wizard to define a property capturing this requirement.

In the first step, the property wizard asks questions to determine the property's scope. The questions and answers are shown in Figure 25.

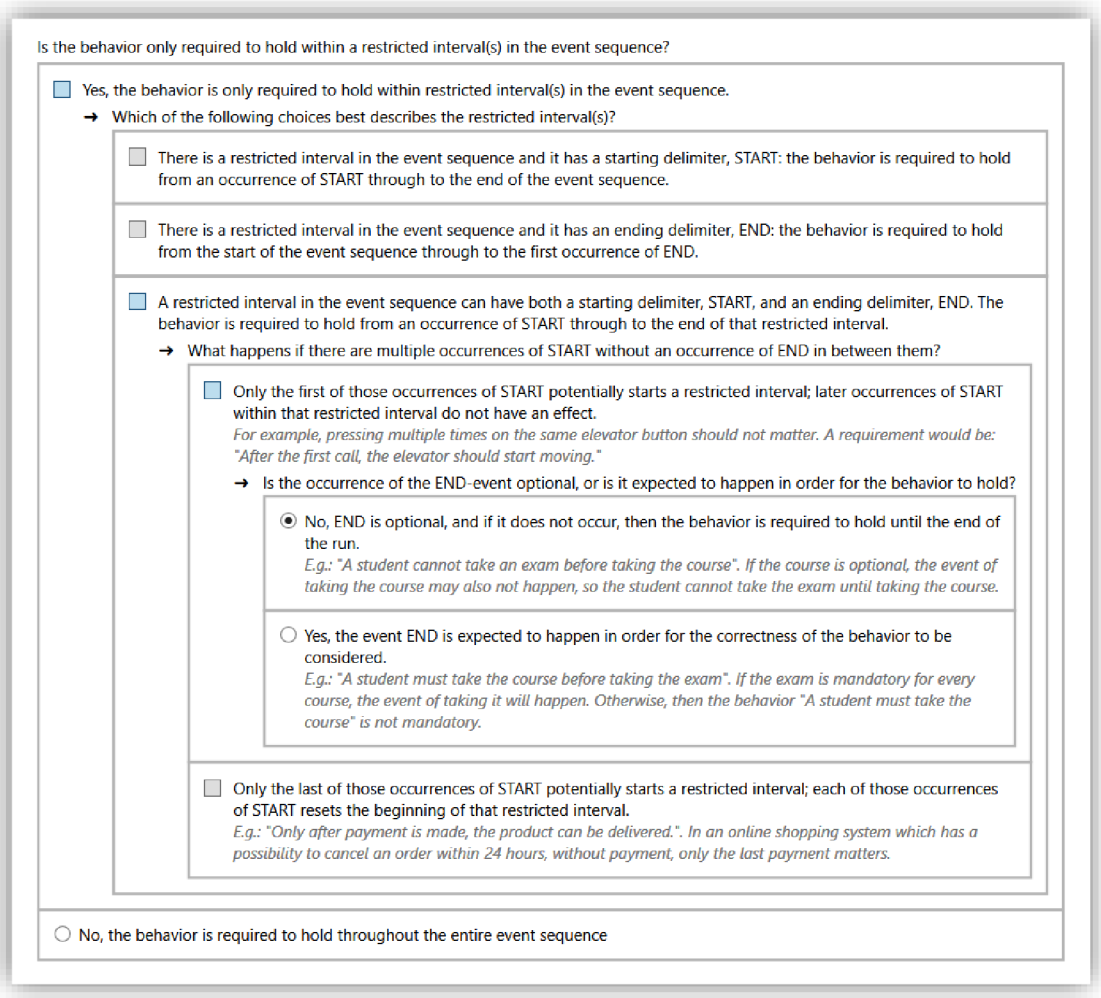

Figure 25 Property Scope page of our Temporal Property Specification Wizard

In the second step, the property wizard asks questions to determine the property behaviour. The questions and answers are shown in Figure 26.

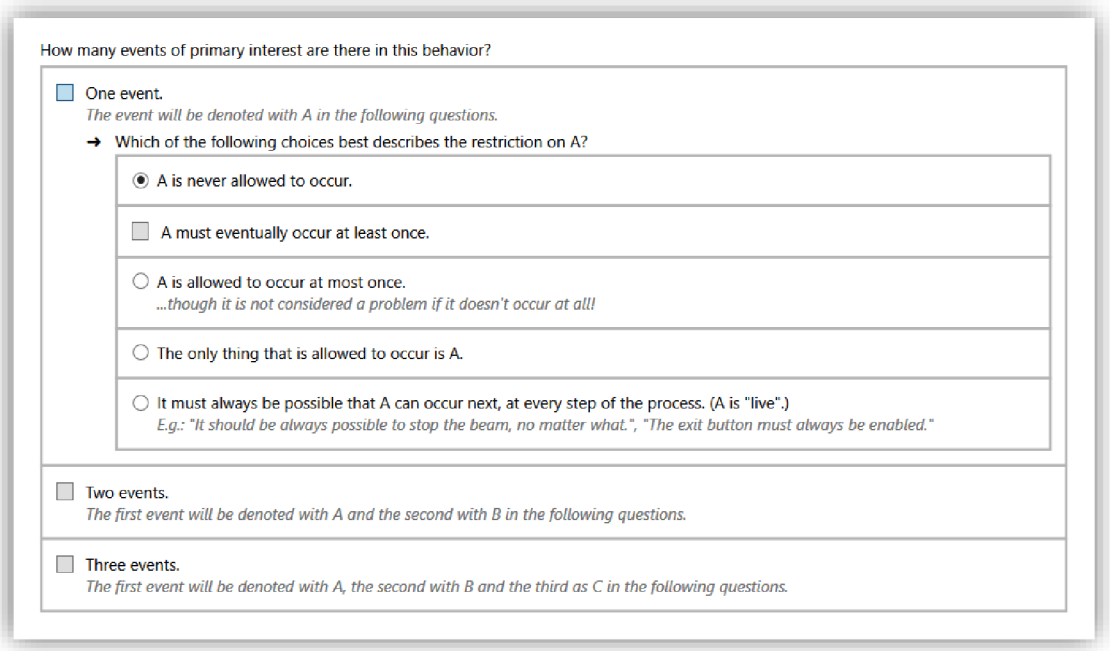

Figure 26 Property Behavior page of our Temporal Property Specification Wizard

In the third and last step, the property wizard asks for the events in the property pattern that results from the previous answers. The questions and answers are shown in Figure 27.

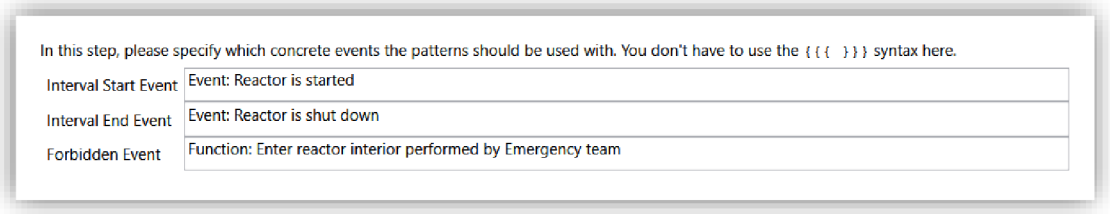

Figure 27 Event Specification page of our Temporal Property Specification Wizard

The Temporal Property Specification Wizard generates the property given in Listing 41. While the property can easily be expressed in natural language, the resulting formalization is shown to be complex and hard to read and understand.

```
[true*. ( {{{Event: Reactor is started}}} ) . (not ( {{{Event: Reactor is shut
down}}} ) )*. ( {{{Function: Enter reactor interior performed by Emergency team}}}
) ] false
```

Listing 41 Exemplary property generated by the Temporal Property Specification Wizard

Checking if the generated property is fulfilled by our exemplary model yields the result that the property is not fulfilled, i.e. the model entails a situation in which the emergency team enters

the reactor while it is running. Inspecting the counterexample information allows us to find the elements involved in this situation. The counterexample stack is given in Figure 28.

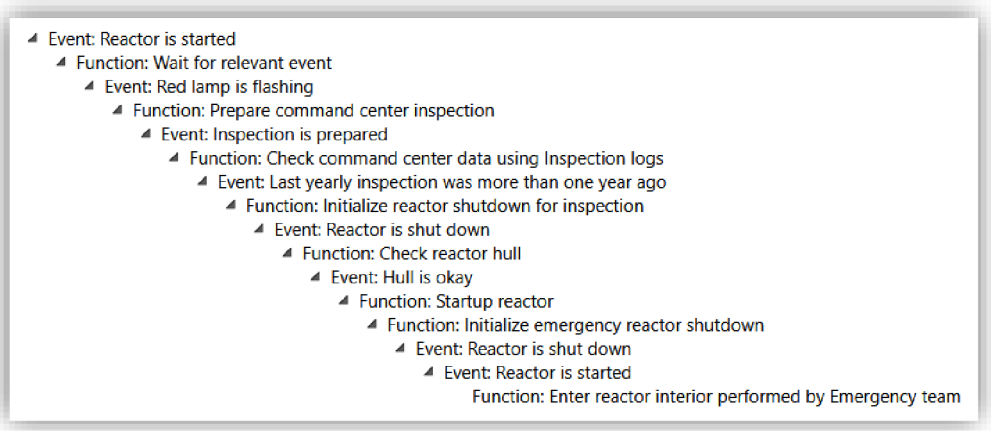

Figure 28 Counterexample stack for the second case study

The counterexample stack implies that a situation can happen in which the reactor is shut down, then directly started again, and then entered by the emergency team. Double-clicking on the last two events allows finding the element occurrences that reported the events. In Figure 29, we give sections of the process model with the highlighted elements that the second to last event (a), and the last event (b) was reported for.

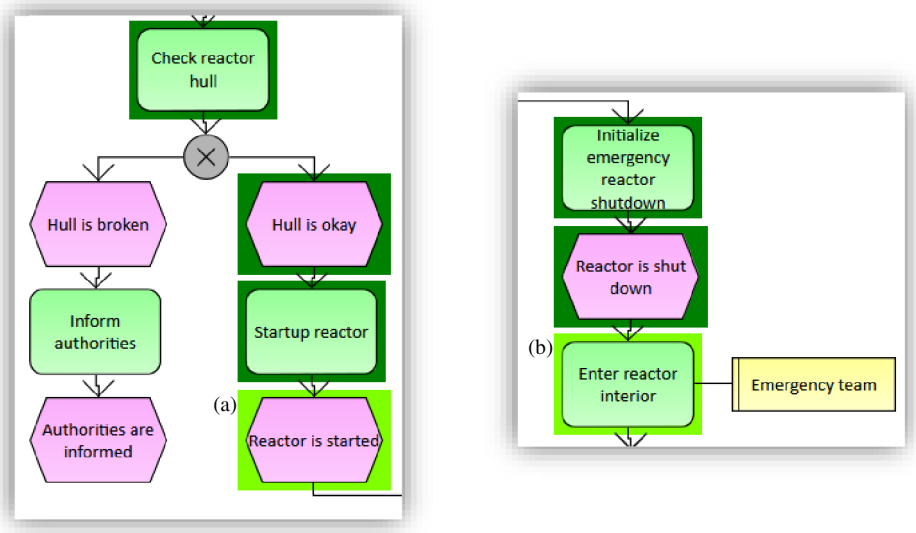

Figure 29 Highlighted elements relevant for the counterexample of the second case study

From this information we can deduce: If a weekly inspection happens to occur when reactor is in emergency mode, the emergency team might enter a started reactor. Careful inspection of the process model would have yielded the same result. We nevertheless hope that our case study shows that a situation like this can easily be overlooked already in a comparatively small model.

7.3 Case Study 3

In practice, models may be interlinked and not all models may use the same modeling languages. Using our third case study, we illustrate that our implementation supports cross-model and cross-language analysis. We further show that it supports events reported for Relationships, not just for Objects. Finally, we show how our implementation visualizes counterexample information that contain "jumps" from one model to another.

We introduce HSEPCwI, an extension of HSEPC with the additional object type *Interface* that allows to be refined by a Model. In Figure 30 we show a section of the model of the previous case study where the "Check reactor interior" *Function* is replaced with a corresponding *Interface*. The highlighting can be ignored for now; we cover it further below.

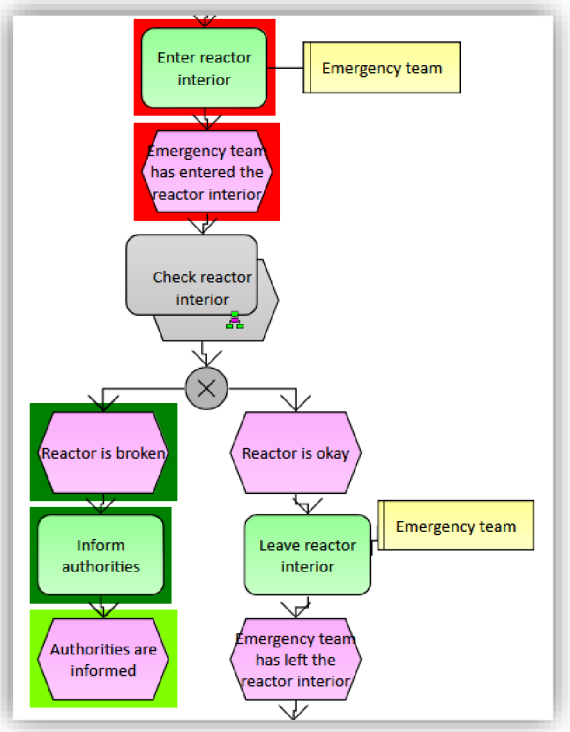

Figure 30 Section of an exemplary HSEPCwI model with highlighted elements

When enabling an *Interface* without refinements, its outgoing relationships will be enabled. When enabling an *Interface* with refinements, it will enable the refinement Model in such a way that the refinement Model can "jump back" to the refined *Interface* and pass data to it. The Interface then passes this data on to its outgoing relationships. In HSEPCwI, the semantic of *XOR* is slightly different than in HSEPC: If an *XOR* is enabled with passed data, it will only enable outgoing relationships that lead to objects whose captions match this passed data. It does not pass on the data any further.

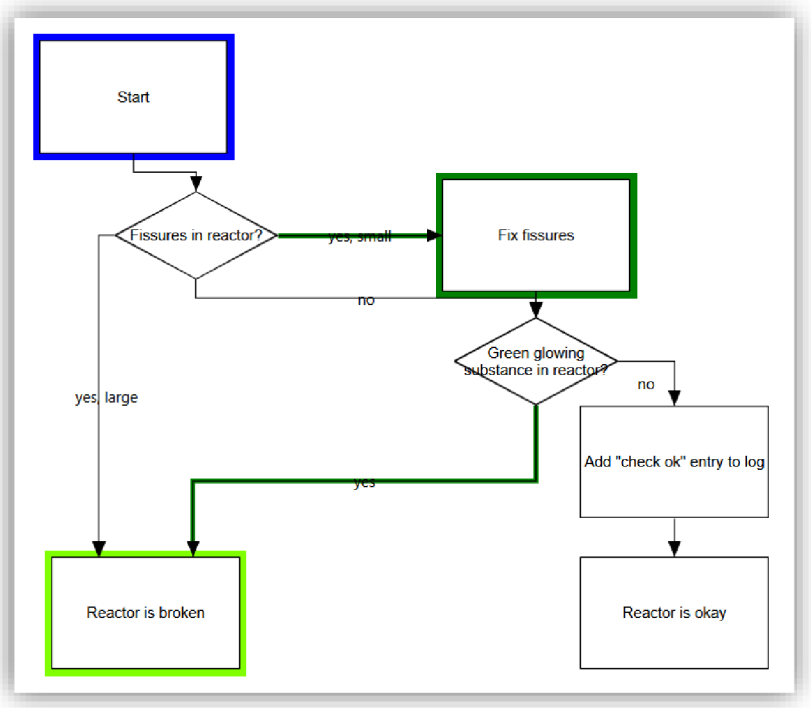

Figure 31 An exemplary SDTL model with highlighted elements

We introduce our Simple Decision Tree Language (SDTL). It has three element types: the object types *State* (represented as a box) and *Decision* (represented as a rhombus), and a generic relationship type (represented as an arrow) that connects any object with another one. In Figure 31, we give an exemplary SDTL model that refines the *Interface* object in the HSEPCwI model of this case study. Again, the highlighting in the Figure can be ignored for now.

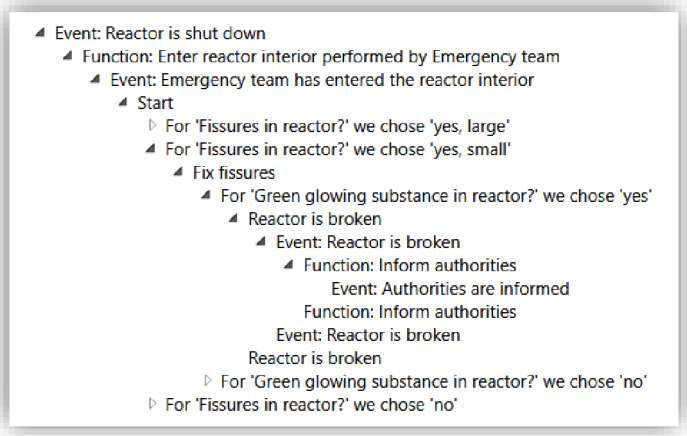

Figure 32 Fragment of the witness information of case study 3

The decision tree language starts with the first found state without incoming relationships, goes through decision tree, interpreting each *Decision* as an exclusive choice. Once reaching a final

state, i.e. a state without outgoing relationships, it enables the object the SDTL model was enabled from, passing the caption of the final state.

For our case study, we want to check if finding green glowing substance in reactor always leads to informing the authorities. Using our approach, we find out that this property is fulfilled, i.e. whenever green glowing substance is found, the authorities will be informed. We give a fragment of the resulting witness information in Figure 32.

The highlighting in the SDTL model given in Figure 31 reflects the result of double-clicking on a “Reactor broken” entry in the witness event stack of Figure 32. The highlighting in the HSEPCwI model given in Figure 30 reflects the result of double-clicking on the “Event: Authorities are informed” entry. By clicking on events in the event stacks that were reported for elements in different models, our implementation allows a user to quickly jump between multiple models containing relevant elements.

As also shown in Figure 30, the color of highlighting changes between the HSEPCwI elements that reported events before and after the switch to the refinement model. This demonstrates that our implementation allows a user to easily pinpoint where a “jump” to another model took place when analyzing the counterexample or witness information for some model.

8 Discussion, Outlook and Summary

In this chapter, we discuss our work's results by comparing our approach with other process model analysis approaches and by identifying obstacles for the practical applicability of our approach. We further outline concepts how our approach could be extended and enhanced and finally give a summary of our work.

In the first section, we compare our model checking-based process model analysis approach with a class of other model analysis approaches that are not restricted to a specific modeling language. In the second section, we discuss the applicability of our approach in practice and identify potentials for follow-up work. In the third section, we summarize our work.

8.1 Comparison with Model Structure-Based Process Model Analysis Approaches

Besides our approach as presented in this thesis, there is only one class of process model analysis approaches known to us that is not restricted to a specific modeling language. Approaches in this class allow formulating and checking for properties formulated over the structure of a model. We therefore call these approaches "structure-based". In this section, we compare our model checking-based process model analysis approach with structure-based approaches.

One structure-based model analysis approach is based on formulating a property as pattern as a tree of operations over sets of model elements. The model's fulfilment of the property is determined by evaluating the operations of the tree and checking if a pattern match was found this way. (Delfmann et al. 2010; Delfmann, Steinhorst, et al. 2015) Another such structure-based approach is based on the subgraph isomorphism problem: A "haystack" graph is derived from a business process model's structure. A property is formulated as a "needle" graph pattern. The model's fulfilment of the property is determined by determining whether the haystack graph contains a subgraph that is isomorphic to the needle graph pattern. (Delfmann, Breuker, et al. 2015)

The core difference between model structure-based analysis approaches and our model checking-based analysis approach lies in the aspects that input properties can describe: In structure-based approaches, properties describe aspects of the structure of process models. In our model checking-based approaches, properties describe aspects of the meaning of process models, i.e. the events entailed by the models' execution semantics.

Most process modeling languages have their own symbols and their own syntax. Therefore, the structure of a model for the same set of business processes generally differs between languages. When formulating properties for structure-based approaches, a property must be specified for a set of anticipated process modeling languages with their respective symbols and syntaxes. If

properties should be used to reason about the business processes entailed by some model, the execution semantics of the anticipated process modeling languages must be implicitly captured in each property. In contrast, our model checking-based approach requires the execution semantics to be explicitly specified only once upfront for the required languages; properties can then directly describe aspects about the events entailed by process models in languages that semantics were specified for.

We give some user advantages and disadvantages of our model checking-based approach compared to the model structure-based approaches.

Property Specification Effort – Directness of Property Specification Workflow: Specifying properties using our approach is arguably more direct, requiring less property specification effort: A user can directly specify sequences of forbidden or required events using our approach. In structure-based approaches on the other handy, an intermediate step is usually required. In this step, a suitable specification must be found for the sequence of forbidden of required events that corresponds to a searchable part of the model's structure. We show with an example that finding such a specification can be difficult.

Take a property that requires an event B to occur at most once in business processes. Using a structure-based approach, a naïve specification for such a property could for example be to search for an object corresponding to event B and to search for a directed outgoing path from this object to another object that also corresponds to event B. The property is fulfilled if no such path is found.

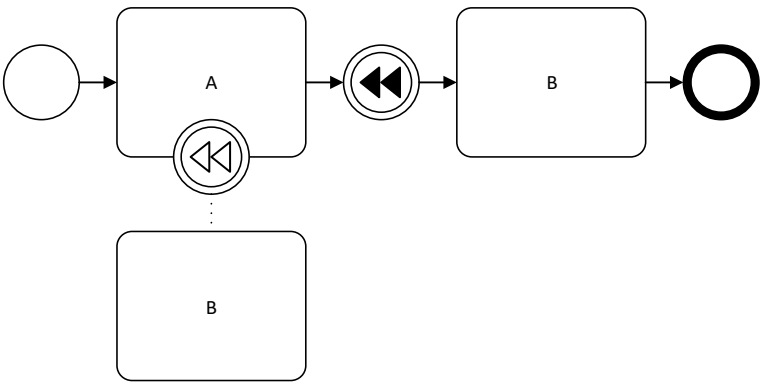

Figure 33 BPMN model with compensations

Such a specification is however incomplete as it would not generally report the correct result. For example, it the specified structure could not be found in the BPMN model in Figure 33, assuming each activity corresponds to an event as specified by the respective activity's label. The model does however entail a process containing two B events: The intermediate throwing compensation event (the circle with the two black left-pointing arrows) enables the activity with label B that is attached to the compensation boundary event (the circle with the two white left-

pointing arrows), followed by enabling the second activity with label B in the normal process flow.

BPMN has several further constructs where the execution semantics does not correspond closely to a path through the model structure. Therefore, finding suitable property specifications that correspond to parts of the model structure is arguably difficult, and may even be impossible for some properties.

Property Specification Effort – Reusability of Property Specifications: If execution semantics for different process modeling languages are appropriately specified for our approach, then properties can be specified independently from the process modeling language that the models to analyze are created in. A user can re-use such a language-independent property for analysis of models in different languages. In structure-based approaches the same property might have to be specified differently for different modeling languages, causing more property specification effort for the user.

Property Specification Effort – Reusability of Execution Semantics Specifications: As established further above, execution semantics must be captured by every property in structure-based approaches. In our approach, the execution semantics needs to be explicitly specified only once. Depending on the complexity of the process modeling language's semantic and on the analysis goals, the effort to specify execution semantics and temporal properties can however still be higher than defining a set of structural patterns. With our approach, economics of scale may be realized if many different properties are to be analyzed for models of the same languages. In a setting where the number of modeling languages involved in process model analysis is mostly constant and low, and the number of involved properties is higher, we expect the initial effort to specify execution semantics to pay off, compared to finding suitable model structure specifications for each property.

Understandability of Property Specifications: Property specifications for our approach may be less understandable for a user, compared to properties for subgraph isomorphism-based approaches that use visual representations for structural patterns. Understanding such a visual pattern may arguably be easier than understanding a temporal formula in a complex formulaic language like in our approach or in an approach based on set operations.

Applicability for Different Analysis Goals: Our approach has the goal to enable a user to formulate a temporal property and to check if this property is fulfilled by business processes entailed through execution semantics by a process model. As such, its application area is restricted to fulfilling this goal. Approaches allowing to formulate properties about the structure of models may have additional application areas. On this basis, structure-based approaches can be considered as arguably being more versatile.

Theoretical Computational Complexity: Advantages of the different approaches w.r.t. computational complexity depend on the given input model, its underlying execution semantics, and the given property. The model checking algorithm employed in our work has a space and time complexity linear in the number of operators in the formula and the size of the LTS. (CADP manual authors 2017f, sec. Model Checking Complexity) Subgraph isomorphism was shown to be NP-complete. (Delfmann, Breuker, et al. 2015, p. 477) From a theoretical point of view we expect our approach to work more efficient than subgraph isomorphism-based approaches, reducing the user's waiting times for results. We were not able to find information on the space and time complexity of the set operation-based approach and can therefore not compare it with our approach w.r.t. algorithmic efficiency. Apart from the theoretical computational complexity, it may be interesting to compare how fast and with how much memory usage the different approaches perform in practice.

8.2 Discussion and Outlook on Our Process Model Analysis Approach

In this section, we derive and identify potentials for future research and for improving and enhancing our work by discussing the applicability of our approach and of our implementation.

In the first subsection, we present potentials that can make implementations of our approach more robust or faster. In the second subsection, we present potentials to extend the functionality of an implementation of our approach. In the third subsection, we discuss our approach conceptually and present ideas to improve and test its applicability.

Within each subsection, we give the potentials for future work ordered by our expected realization effort, starting with the potential that we expect to cause the lowest effort if realized.

8.2.1 Non-Functional Improvement Potentials

Making Diagnostics Information Available to Plugin Differently. We rely on parsing of AUT files and of their labels to make CADP's model checking diagnostics information available to our plugin. Hubert Garavel, one of CADP's core maintainers, signalized in unpublished communication that the format of labels generated by CADP in AUT files may be modified for future version of CADP. Such a modification may require adjusting our parsing implementation.

To avoid such adjustments in the long term, it may be interesting to implement a format-independent approach of making diagnostics information available to our plugin. As such an alternative approach, a new stable file format could be defined for transferring relevant aspects of the diagnostics information from the CADP tools to our plugin. A small C-based or C++-based program could be written to create files in this format. This program could make use of

CADP's C API to read a diagnostics BCG file and would write the information in the newly defined format.

Increasing CADP Performance. The CADP software tools being Unix-targeted and [em] being Windows-targeted, the integration of the two toolsets requires difficult solutions that have their downsides. In our implementation we run the CADP tools on Cygwin (Cygwin project home page authors 2017), a Unix-like environment for Windows. Running the CADP tools this way comes with a heavy performance penalty: Processing takes much more time than on a "true" Unix system.

Even with the small models presented in our work, each property validation took several minutes on Cygwin. From the long processing times on Cygwin, we assume that our implementation operates too slowly for real-world scenarios. An implementation that should be applicable to real-world scenarios would need to find a way to reduce processing times dramatically. For some examples, we used a Linux system to test running the same CADP commands as our plugin does. Model checking completed within only a few seconds on Linux for the tested examples.

This indicates that different way of implementing our approach could increase performance. For example, a network-based communication protocol could be designed that allows exchanging relevant data between an [em] plugin (running on Windows) and a server application (running on a Unix-based system). The server application could perform model checking operations using CADP, and send back the results to the [em] plugin. If there was a requirement to still run both applications on the same physical computer, a virtual Unix machine for the server application could be set up on a Windows system for [em] – or the other way around.

Another alternative may be to make the CADP software tools available for the Windows Subsystem for Linux (WSL) (Windows Subsystem for Linux Documentation contributors 2016). According to unpublished communication from Hubert Garavel, one of the core CADP maintainers, the CADP team initially found WSL a promising way to allow for fast model checking with CADP on Windows, but finally found the effort to get it running on WSL to be too high.

Generalizing Approach to Use Different Model Checkers. Our implementation is based on the CADP tools and tightly coupled with them. It may be interesting to realize our approach with different model checkers or to develop a generalized implementation that could easily be extended with different model checkers. The different implementations and model checkers could then be compared w.r.t. performance and versatility.

One especially interesting model checker for such an implementation is mCRL2. As of writing this thesis, mCRL2's witness or counterexample information do not allow to draw conclusions about the model elements that are "responsible" for a temporal property to be fulfilled or not fulfilled.

An approach based on mCRL2's event reachability analysis capability can however generate counterexample or witness information that could be used for our approach. In this approach temporal properties are translated into "observer" or "monitor" processes. Such a monitor process is integrated to a formal process specification and monitors all the reported events. It will report a special event if it detects a sequence of events that allows determining the fulfillment of the property. Using mCRL2's event reachability analysis allows finding a sequence of events that lead to the special monitor process event being reported. This approach does however only work for a limited subset of temporal properties. (Remenska 2016, chap. 5.4.4; Remenska et al. 2014)

8.2.2 Functionality-Extending Improvement Potentials

Introducing Value Object Support. As established in section 4.1, we did not implement support for the full [em] data model to keep the demonstration of our approach simple: Values carried by *Objects* are not available in our implementation. If model checking with value-carrying *Objects* turns out to be required in practice, support for such values can easily be added to our implementation.

Introducing Reusable Formulaic Expressions and Behaviors. Typical programming languages offer the concepts of "functions" and "procedures", allowing to abstract code that is used more than once. Our implementation does not support such concepts to abstract repetitions in formulaic expressions or behavior sequences. Extending our approach with such concepts may improve the practical usability of our approach and the conciseness of behavior sequences as well as formulaic expressions.

Introducing Syntax Highlighting in Formulaic Expressions. Modern integrated development environments support reviewing code through coloring different syntactic constructs in different colors. Extending our approach with such syntax highlighting may improve the practical usability of an implementation of our approach.

Extending Event Data Types. Our current implementation only allows to use strings as public events. It may be useful to allow further data types in public events. For example, a "Invoice paid" event could additionally carry the amount that was paid in a structured way. Our implementation requires to encode such information in the event string, e.g. as "Invoice paid (Amount: 100 Euro)". While the data type string is versatile in this regard, temporal property specifications can arguably become inconvenient. A property requiring the occurrence of an

invoice payment event for any amount would require a suitable event string pattern specification that does not take the amount into consideration. A property requiring the occurrence of an invoice payment event for an amount larger than some value would even require a string parsing mechanism that could interpret the string-encoded amount value in the event string.

The CADP and mCRL2 tools allow events to carry values of different data types. To different extends, both solutions also allow specifying data-rich temporal properties, i.e. properties that take the values carried by the events into consideration. Extending our approach with the ability to report events carrying additional data in different types may make the specification of properties easier in settings with data. Extending our approach further to allow a user to specify more than the two pre-defined custom data types may further increase the applicability of our approach.

Enabling Analysis with Infinite Data Types. The CADP tools work under the assumption of finite data types. For example, integer values in our implementation are 16-bit values, i.e. only integers in the range −32,768 through 32,767 can be used in formal processes defined with our implementation. While this range may arguably be sufficient for many practical problems, there may be formal processes requiring larger and possibly infinite data type ranges. Such processes cannot meaningfully be checked with CADP.

Model checking such a process requires different approaches. Translating a formal process specification not into a LTS but into a Symbolic Transition System was proposed as such an approach. (Calder et al. 2001; Calder and Shankland 2001) Using a first-order extension of μ-calculus was proposed as another one. (Groote and Mateescu 1999) We assume that such approaches need to be adopted for our approach to handle process model analysis problems requiring more complex data types.

Making Counterexample Information More Tangible. In subsections 7.1.3 and 7.2.2, we demonstrated that counterexample or witness information can be helpful for understanding why a property is fulfilled or not fulfilled. We assume however that counterexample or witness information may be more complex and hard to understand when applying our approach on real-world business process models that especially handle much variable data. In the context of model checking for computer program code, tools were developed to make understanding counterexample and witness information easier and more tangible.

For example, one such tool can produce a variation of counterexample information that is close to the original counterexample but does not violate the fulfilment of a property. (Groce et al. 2004) Another tool allows to find test vectors for a target predicate, i.e. tuples of input values for a program that lead to a predicate to be satisfied. Using these input values, the dynamic behavior of the program can be studied to gain further insight into reasons for the predicate to

be satisfied. (Beyer et al. 2004) It may be interesting to adopt such approaches for the context of model checking for business process models.

8.2.3 Conceptual Future Work

Improving Assistance in and Accessibility of Temporal Property Specification. In section 7.2.2, we demonstrated how a temporal property can be generated with the Temporal Property Specification Wizard. Despite being a property with a simple natural language specification, its formal specification is complex and difficult to understand.

We expect the complexity of temporal property specifications to be barely manageable in more complex scenarios that might be required in practice, and we assume that most process model analysis practitioners do not have deep knowledge around temporal property specification. We argue that it is a difficult task to convey knowledge required to understand complex temporal properties in languages inspired by or based on the μ-calculus. We assume that most practitioners would not accept an analysis approach that requires such deep formal knowledge to produce results that are of practical use.

On this basis, we assume that strong assistance in specifying temporal properties is a necessity to ensure acceptance for a model checking-based business process analysis approach. While the Temporal Property Specification Wizard in our implementation is providing helpful assistance already, we assume that further assistance is required for more complex property requirements.

Another way to make temporal properties more accessible may be the introduction of a visual notation for them. An example for such a visual notation is an BPMN-inspired one that was proposed for LTL. (Brambilla 2005) In the context of model checking for computer programs, another visual notation was proposed that was inspired by UML sequence diagrams. (Remenska 2016, chap. 5.4.3) It may be interesting to design a visual notation for MCL or similar temporal specification languages based on the μ-calculus and to implement a functionality that allows a user to specify or analyze properties in such a visual notation.

Testing Applicability of Approach. In section 2.1, we defined a “business process” using a simplification that stripped away any relevance towards a business need. For the perspective of our work, we could use this simplification. However, with process models typically being defined with a business need in mind, we assume that business process analysis should usually help to fulfill a business need. On this basis we assume that process model analysis is only helpful if conclusions drawn from an analysis can be translated back into the reality that the models and their execution semantics should describe.

From this argument, another aspect of our approach can be derived that may be an obstacle for practical application of our approach: the mutual dependency between models with their

content, specifications of formal execution semantics, and specifications of properties. We explain this triangular dependency in pairs:

a) Execution semantics specifications and property specifications may not fit together, for example because the events entailed through execution semantics have a different representation than expected in temporal properties. For illustration, take a temporal property that requires an event "Activity 'Pay Invoice' performed" to occur, and assume that the execution semantics entails only the congruent but differently represented event "Pay Invoice function executed", i.e. an event with a different string representation than used in the property.

b) Execution semantics specifications and models with their content may not fit together, for example because execution semantics were specified under specific expectations about models and their content, but models do not meet these expectations. For illustration, assume that execution semantics is specified under the assumption that each activity has an assigned responsible person, but some models have activities without such a person.

c) Property specifications and models with their content may not fit together, for example because properties refer to events that do not have corresponding elements in models. This may be a result of models' low granularity. For illustration, take a temporal property that requires an event "Activity 'Login to Online Banking System' performed" to occur, and a model with an event "Activity 'Pay Invoice' performed". Assume that paying the invoice implicitly requires logging in to the Online Banking system without the model explicitly capturing this implicit relationship.

While the temporal property in the illustration for a) and c) is fulfilled on a conceptual level, the misalignment between execution semantics, the property specifications, and models with their content would make model checking yield a result that indicates non-fulfillment. In the illustration for b), model checking may even yield an undefined result or an error. These illustrations demonstrate that one of the three elements not being aligned with the other two causes the model checking result to not be helpful for a business process model analyst. For our approach to be applicable to real-world business process model analysis problems, we expect aligning these three elements to be a challenge.

In this mutual dependency, we expect real-world models to be particularly challenging for our approach's practical applicability: While in much literature the elicitation and representation of reality in the course of modeling is regarded as largely unproblematic, it was argued that representations of the reality like business process models are generally perspectival simplifications with inherent limits. (Riemer et al. 2013) A possible implication of this argument is: Business process models may not generally capture all pieces of information that are required for meaningful analysis.

It may therefore be interesting to test the applicability of our approach in real-world scenarios. It may also be interesting to compare the real-world applicability of different analysis approaches.

8.3 Summary

We presented a business process model analysis approach based on model checking. We introduced languages for describing a process model's execution semantics at the level of its modeling language by formulating behavior sequences and assigning them to models and their elements on the meta-level.

Given a model, its meta model, given execution semantics described on the model's meta-level, and given some temporal property, our approach uses model checking algorithms to determine if the model fulfills the temporal property and to get a hint why the property is fulfilled or why it is not, respectively.

We surveyed model checkers that an implementation of our approach could be based on and selected the *Construction and Analysis of Distributed Processes* (CADP) model checker as the base of our implementation. We developed an approach to translate [em] data and assigned behavior sequences into process specifications that can be processed by the CADP model checker. We developed a macro extension for the CADP-supported temporal property specification language MCL that allows a user to formulate properties easily while keeping the possibility of mapping events from counterexample or witness information back to model elements that they were reported for.

We collected requirements that an implementation of our approach must fulfill to be usable for a user. We developed a plugin for [em] that implements our approach. Finally, using our [em] plugin, we demonstrated in three case studies how our implementation can be used in practice.

We discussed the applicability of our approach and our implementation and named various ways how they can be extended and enhanced. While our approach proved to work in our exemplary artificial case studies, we also identified obstacles for the practical applicability of our approach. It would therefore be interesting to test in future work if our approach is applicable to real-world business process model analysis problems.

References

The URLs in this bibliography were all valid when last accessed on 2017-12-10.

.NET Docs contributors 2017. ".NET Regular Expressions Reference," in *.NET Docs*. Available at: https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions.

van der Aalst, W. M. P., Desel, J., and Kindler, E. 2002. "On the semantics of EPCs: A vicious circle," in *Proceedings of the EPK 2002: Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten*, M. Rump and F.J. Nuttgens (eds.), Trier, Germany: Gesellschaft für Informatik, pp. 71–80.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. 2003. "Workflow Patterns," *Distributed and Parallel Databases* (14:1), pp. 5–51.

van der Aalst, W. M. P., and ter Hofstede, A. 2017. "Workflow Patterns Home Page." Available at: http://www.workflowpatterns.com/.

American National Standard for Information Standards 1986. *Coded Character Sets — 7-Bit American Standard Code for Information Interchange (7-Bit ASCII) ANSI X3.4-1986*.

Arsac, W., Compagna, L., Pellegrino, G., and Ponta, S. E. 2011. "Security Validation of Business Processes via Model Checking," *Lecture Notes in Computer Science* (6542), pp. 29–42.

Becker, J., Breuker, D., Weiß, B., and Winkelmann, A. 2010. "Exploring the Status Quo of Business Process Modelling Languages in the Banking Sector – An Empirical Insight into The Usage of Methods in Banks," *ACIS 2010 Proceedings*, p. Paper 8.

Becker, J., Delfmann, P., Dietrich, H.-A., Steinhorst, M., and Eggert, M. 2014. "Business process compliance checking - applying and evaluating a generic pattern matching approach for conceptual models in the financial sector," *Information Systems Frontiers* (17), pp. 1–47.

Becker, J., Delfmann, P., and Knackstedt, R. 2004. "Konstruktion von Referenzmodellierungssprachen - Ein Ordnungsrahmen zur Spezifikation von Adaptionsmechanismen für Informationsmodelle," *Wirtschaftsinformatik* (46:4), pp. 251–264.

Becker, J., and Schütte, R. 2004. *Handelsinformationssysteme*, MI Wirtschaftsbuch.

Bergstra, J. A., and Klop, J. W. 1984. "Process Algebra for Synchronous Communication," *Information and Control* (60:1–3), pp. 109–137.

Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., and Majumdar, R. 2004. "Generating Tests from Counterexamples," in *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, United Kingdom: ACM, pp. 326–335.

Bolognesi, T., and Brinksma, E. 1987. "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems* (14:1), pp. 25–59.

Brambilla, M. 2005. *LTL Formalization of BPML Semantics and Visual Notation for Linear Temporal Logic*.

CADP manual authors 2017a. "AUT manual page," in *CADP website*. Available at: http://cadp.inria.fr/man/aut.html.

CADP manual authors 2017b. "BCG_WRITE manual page," in *CADP website*. Available at: http://cadp.inria.fr/man/bcg_write.html.

CADP manual authors 2017c. "BCG manual page," in *CADP website*. Available at: http://cadp.inria.fr/man/bcg.html.

CADP manual authors 2017d. "BISIMULATOR manual page," in *CADP website*. Available at: http://cadp.inria.fr/man/bisimulator.html.

CADP manual authors 2017e. "EVALUATOR4 manual page," in *CADP website*. Available at: http://cadp.inria.fr/man/evaluator4.html.

CADP manual authors 2017f. "MCL manual page," in *CADP website*. Available at: http://cadp.inria.fr/man/mcl.html.

CADP manual authors 2017g. "REDUCTOR manual page," in *CADP website*. Available at: http://cadp.inria.fr/man/reductor.html.

CADP manual authors 2017h. "regexp manual page," in *CADP website*. Available at: http://cadp.inria.fr/man/regexp.html.

Calder, M., Maharaj, S., and Shankland, C. 2001. "An Adequate Logic for Full LOTOS," in *Proceedings of the 10th International Symposium of Formal Methods Europe*, Berlin, Germany: Springer, pp. 384–394.

Calder, M., and Shankland, C. 2001. "A Symbolic Semantics and Bisimulation for Full LOTOS," in *Proceedings of the 21st International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2001)*, pp. 184–200.

Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Lang, F., McKinty, C., et al. 2017. *Reference manual of the LNT to LOTOS translator (version 6.7)*, INRIA.

Clarke, E. M., and Emerson, E. A. 1981. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," *Lecture Notes in Computer Science* (131).

Clarke, E. M. 2008. "The Birth of Model Checking," *Lecture Notes in Computer Science* (5000), pp. 1–8.

Cygwin project home page authors 2017. "Cygwin project home page." Available at: https://cygwin.com/.

Davenport, T. H. 1993. *Process Innovation: Reengineering Work Through Information Technology*, Harvard Business School Press.

Delfmann, P., Breuker, D., Matzner, M., and Becker, J. 2015. “Supporting Information Systems Analysis Through Conceptual Model Query - The Diagramed Model Query Language (DMQL),” *Communications of the Association for Information Systems* (37), pp. 473–509.

Delfmann, P., Herwig, S., Karow, M., and Lis, Ł. 2008. “Ein konfiguratives Metamodellierungswerkzeug,” *Proceedings of the Workshops Colocated with the MobIS2008 Conference: Including EPK2008, KobAS2008 and ModKollGP2008*, pp. 109–127.

Delfmann, P., and Hübers, M. 2015. “Towards Supporting Business Process Compliance Checking with Compliance Pattern Catalogues,” *Enterprise Modelling and Information Systems Architectures* (10:1), pp. 67–88.

Delfmann, P., Sebastian, H., Lis, Ł., Stein, A., Tent, K., and Becker, J. 2010. “Pattern Specification and Matching in Conceptual Models,” *Enterprise Modelling and Information Systems Architectures* (5:3), pp. 24–43.

Delfmann, P., Steinhorst, M., Dietrich, H. A., and Becker, J. 2015. “The generic model query language GMQL - Conceptual specification, implementation, and runtime evaluation,” *Information Systems* (47), pp. 129–177.

van Dongen, B. F., Jansen-Vullers, M. H., Verbeek, H. M. W., and van der Aalst, W. M. P. 2007. “Verification of the SAP reference models using EPC reduction, state-space analysis, and invariants,” *Computers in Industry* (58:6), pp. 578–601.

Emerson, E. A., and Lei, C.-L. 1986. “Efficient Model Checking in Fragements of the Propositional Mu-Calculus (Extended Abstract),” in *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*, Cambridge, MA, USA: IEEE Computer Society Press, pp. 267–278.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Groce, A., Kroening, D., and Lerda, F. 2004. “Understanding Counterexamples with explain,” *Computer Aided Verification* (3114), pp. 318–321.

Groote, J. F., and Mateescu, R. 1999. “Verification of Temporal Properties of Processes in a Setting with Data,” *Lecture Notes in Computer Science* (1548), pp. 74–90.

Hammer, M., and Champy, J. 1993. *Reengineering the Corporation: A Manifesto for Business Revolution*, Harper Business.

Hoare, C. A. R. 1978. “Communicating Sequential Processes,” *Communications of the ACM* (21:8), pp. 666–677.

Hoare, C. A. R. 1980. “A Model for Communicating Sequential Processes,” in *On the Construction of Programs*, R.M. McKeag and A.M. McNaghton (eds.), London, United Kingdom; New York, United States of America: Cambridge University Press, pp. 229–243.

International Organization for Standardization 1989. *Binary floating-point arithmetic for microprocessor systems (ISO/IEC 559:1989).*

International Organization for Standardization 2011. *Information technology – Programming languages – C (ISO/IEC 9899:2011)*.

International Organization for Standardization 1996. *Information technology – Syntactic metalanguage – Extended BNF (ISO 14977:1996)*.

International Union of Railways (UIC), Railsafe Consulting Ltd., University of York, University of Southampton, and Laboratory for Quality Software (LaQuSo) 2009. *INESS_WS D_Deliverable D.4.1_Documented strategy for Verification and Validation_Report*.

Keller, G., Nüttgens, M., and Scheer, A.-W. 1992. "Semantische Prozeßmodellierung auf der Grundlage 'Ereignisgesteuerter Prozeßketten (EPK),'" *Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), Universität des Saarlandes* (89).

Kozen, D. 1982. "Results on the Propositional μ-Calculus," in *Proceedings of the Special Issue 9th International Colloquium on Automata, Languages and Programming*, Aarhus, Denmark: Elsevier, pp. 333–354.

Lahtinen, J., Valkonen, J., Björkman, K., Frits, J., Niemelä, I., and Heljanko, K. 2012. "Model checking of safety-critical software in the nuclear engineering domain," *Reliability Engineering and System Safety* (105), pp. 104–113.

Lamport, L. 1977. "Proving the Correctness of Multiprocess Programs," *IEEE Transactions on Software Engineering* (SE-3:2), pp. 125–143.

Mateescu, R., and Thivolle, D. 2008. "A Model Checking Language for Concurrent Value-Passing Systems," in *Proceedings of the 15th International Symposium on Formal Methods*, Turku, Finland: Springer, pp. 148–164.

Mendling, J. 2007. "Detection and Prediction of Errors in EPC Business Process Models."

Microsoft Corporation 2017a. "How to: Create and customize a web app in Access," in *MSDN Library*. Available at: https://msdn.microsoft.com/en-us/library/jj249372.aspx.

Microsoft Corporation 2017b. "System.Double.Parse Method," in *.NET Framework documentation*. Available at: https://msdn.microsoft.com/en-us/library/fd84bdyt(v=vs.110).aspx.

Microsoft Corporation 2017c. "System.Int32.Parse Method," in *.NET Framework documentation*. Available at: https://msdn.microsoft.com/en-us/library/b3h1hf19(v=vs.110).aspx.

Milner, R. 1980. *A Calculus of Communicating Systems*, University of Edinburgh. Department of Computer Science. Laboratory for Foundations of Computer Science.

Object Management Group 2011. *Business Process Model and Notation (BPMN) Version 2.0*.

Object Management Group 2015. *OMG Unified Modeling Language (OMG UML) Version 2.5*.

Pnueli, A. 1977. “The Temporal Logic of Programs,” in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, Providence, Rhode Island, United States of America: IEEE, pp. 46–57.

Pribnow, H. 2016a. “How to pick any element of a list (and optionally delete it),” in *CADP forums*. Available at: http://cadp.forumotion.com/t443-how-to-pick-any-element-of-a-list-and-optionally-delete-it (Free Registration Required).

Pribnow, H. 2016b. “Parser generator for C# target that enables IntelliSense-like code auto-completion?,” in *StackEchange Community “Software Recommendations.”* Available at: https://softwarerecs.stackexchange.com/questions/30229/parser-generator-for-c-target-that-enables-intellisense-like-code-auto-completi.

Raedts, I., Petkovic, M., Usenko, Y. Y. S., van der Werf, J. M. E. M., Groote, J. F., and Somers, L. J. 2007. “Transformation of BPMN Models for Behaviour Analysis.,” in *Proceedings of the 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS-2007)*, pp. 126–137.

Remenska, D., Willemse, T. A. C., Templon, J., Verstoep, K., and Bal, H. 2014. “Property Specification Made Easy: Harnessing the Power of Model Checking in UML Designs,” in *Lecture Notes in Computer Science*, Berlin, Germany: Springer, pp. 17–32.

Remenska, D. 2016. “Bringing Model Checking Closer To Practical Software Engineering.”

Resnick, M., Kafai, Y., and Maeda, J. 2003. “A Networked, Media-Rich Programming Environment to Enhance Technological Fluency at After-School Centers in Economically-Disadvantaged Communities.”

Riemer, K., Hovorka, D., Johnston, R. B., and Indulska, M. 2013. “Challenging the Philosophical Foundations of Modeling Organizational Reality: The Case of Process Modeling,” in *Thirty Fourth International Conference on Information Systems (ICIS 2013)*.

Smith, J. 2009. “Patterns - WPF Apps With The Model-View-ViewModel Design Pattern,” in *MSDN Magazine Blog*. Available at: https://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

Stirling, C. 1996. “Modal and Temporal Logics for Processes,” *Lecture Notes in Computer Science* (1043), pp. 149–237.

Technische Universiteit Eindhoven 2017. “Introduction to mCRL2,” in *mCRL2 Language Reference*. Available at: http://www.mcrl2.org/web/user_manual/introduction.html.

Wikipedia contributors 2017a. “List of model checking tools,” in *Wikipedia, The Free Encyclopedia*. Available at: https://en.wikipedia.org/w/index.php?title=List_of_model_checking_tools&oldid=795637856.

Wikipedia contributors 2017b. “Nullable type,” in *Wikipedia, The Free Encyclopedia*. Available at: https://en.wikipedia.org/w/index.php?title=Nullable_type&oldid=798046589.

Wikipedia contributors 2017c. “Process modeling,” in *Wikipedia, The Free Encyclopedia*. Available at: https://en.wikipedia.org/w/index.php?title=Process_modeling&oldid=792507909.

Windows Subsystem for Linux Documentation contributors 2016. “Windows Subsystem for Linux Documentation,” in *Microsoft Developer Network*. Available at: https://msdn.microsoft.com/commandline/wsl/about.

Appendix A Semantics of Formulaic Expression Language

In this appendix, we define semantics of our formulaic expression language.

We formally define a set of environments as $Environment = \{(p, m) \mid p \in Environment, m \in M\}$ where p is the "parent" environment, and m is a map from identifiers to values. M is the set of all possible functions that map an identifier to a value, formally $M = \{I_p \to V \mid I_p \in P(I), V \in \bigcup_{T \in Types} T\}$ where P is the function to derive a set's power set, I is the set of all possible identifiers, and $Types = \{Bool, Integer, String, \dots\}$ is the set of all data types supported by the formulaic expression language.

On this basis, we give a recursive definition of a function $eval$ that takes a formulaic expression exp in our language and an environment env, and returns the value that exp evaluates to.

We start with introducing Base token semantics. We denote a `Base` by its type name followed by its bracket-enclosed string value or a bracket-enclosed identifier that plays the role of a placeholder for its string value. The first part of the $eval$ function is defined as follows:

$exp = Boolean(\text{"true"})$	$\to$	$eval(exp, env) = true$
$exp = Boolean(\text{"false"})$	$\to$	$eval(exp, env) = false$
$exp = Integer(r)$	$\to$	$eval(exp, env) = System.Int32.Parse(r)$ where $System.Int32.Parse$ is as defined in (Microsoft Corporation 2017c).
$exp = String(r)$	$\to$	$eval(exp, env) = r_p$ where r_p is r without the first and the last character and where all occurrences of `\"` are replaced with `"` and then all occurrences of `\\` are replaced with `\`.
$exp = Double(r)$	$\to$	$eval(exp, env) = System.Double.Parse(r_s)$ where r_s is r without the first and the last character, and $System.Double.Parse$ is as defined in (Microsoft Corporation 2017b).

To formally specify the evaluation of identifiers, we a helper function called $lookup$. Given an environment and an identifier, this function tries to find and return a mapped value for the

identifier from a search in the given environment and a recursive iteration through the chain of its parent environments. If it does not find the identifier, it returns $null$. We formally define $lookup$ as follows:

$$lookup(env, id) = \begin{cases} env_1(id) & \text{where } id \in Domain(env_1) \\ lookup(env_0, id) & \text{where } id \notin Domain(env_1) \wedge env_0 \neq \emptyset. \\ null & \text{else} \end{cases}$$

We formally specify how to evaluate an `Identifier`. The $eval$ function is defined for `Identifier` as follows:

$$exp = Identifier(r) \quad \rightarrow \quad eval(exp, env) = lookup(env, r).$$

We introduce accessor semantics. To formally specify the evaluation of accessors, we make use of a helper function called $apply$. We give an intuition and then explain parts of its definition.

The $apply$ function is a ternary function taking a value of any data type in our language as its first argument, a PropertyAccessor, FunctionAccessor, or LambdaAccessor as its second argument, and an environment as its third argument. It yields a value of some data type in our language. Informally, $apply$ yields the result of applying an accessor to a value. For example, when provided with a Boolean value b, a PropertyAccessor with the string value `Inverse` and some environment, it yields the inverse (i.e. the negation) of b.

We extend the definition of the $eval$ function with $apply$ to define evaluations of Formulas with Accessors:

$$exp = Formula(f) \text{ “.” } Accessor(a) \quad \rightarrow \quad eval(exp, env) = apply(eval(f, env), a, env)$$

We exemplarily show partial definitions for $apply$ for each of the three Accessor types.

We give a partial $apply$ definition for PropertyAccessors on Boolean values. Let b be a Boolean value different from $null$. Let $pa(id)$ a PropertyAccessor with identifier id. Then:

$$apply(b, pa(\text{“}Inverse\text{”}), env) = \begin{cases} \neg b & if\ b = true\ \vee\ b = false \\ null & if\ b = null \end{cases}$$

$$apply(b, pa(\text{“}AsString\text{”}), env) = \begin{cases} \text{”true”} & if\ b = true \\ \text{”false”} & if\ b = false \\ null & if\ b = null \end{cases}$$

We give a partial $apply$ definition for FunctionAccessors, also on Boolean values. Let $fa(id, args)$ be a FunctionAccessor with some identifier id and a sequence of Formulas $args$ that is derived from recursive iteration of its ArgumentList. Then:

$$apply(b, fa(\text{“}And\text{”}, args), env) = \begin{cases} b \wedge eval(args_0, env) & if\ b = true \vee b = false \\ null & if\ b = null \end{cases}$$

where $|args| = 1$ and $eval(args_0, env)$ is Boolean.

$$apply(b, fa(\text{“}Or\text{”}, args), env) = \begin{cases} b \vee eval(args_0, env) & if\ b = true \vee b = false \\ null & if\ b = null \end{cases}$$

where $|args| = 1$ and $eval(args_0, env)$ is Boolean.

$$apply(b, fa(\text{“}Xor\text{”}, args), env) = \begin{cases} b \oplus eval(args_0, env) & if\ b = true \vee b = false \\ null & if\ b = null \end{cases}$$

where $|args| = 1$ and $eval(args_0, env)$ is Boolean.
$\oplus$ symbolizes the logical operation “exclusive or”.

$$apply(b, fa(\text{“}Equals\text{”}, args), env) = \begin{cases} null & if\ eval(args_0, env) = null \\ true & if\ b = eval(args_0, env) \\ false & if\ b \neq eval(args_0, env) \end{cases}$$

where $|args| = 1$ and $eval(args_0, env)$ is Boolean.

$$apply(b, fa(\text{“}IfElse\text{”}, args), env) = \begin{cases} null & if\ eval(args_0, env) = null \\ eval(args_1, env) & if\ eval(args_0, env) = true \\ eval(args_2, env) & if\ eval(args_0, env) = false \end{cases}$$

where $|args| = 2$ and $eval(args_0, env)$ is Boolean.

We give a partial $apply$ definition for LambdaAccessors on Collection values. Let $la(id, params, exp)$ a LambdaAccessor with identifier id, formula exp, and a sequence of identifiers $params$ that is derived from recursive iteration of its LambdaParameterList. Let c be a Collection (for any type) that is different from $null$. Let $membersOf(c)$ be c’s members. Then:

$$apply\big(c, la(\text{“}All\text{”}, params, exp)\big) = \begin{cases} null & if\ \exists m \in membersOf(c) : eval(exp,\ (env, \{(params_0, m)\})) = null \\ true & if\ \forall m \in membersOf(c) : eval(exp,\ (env, \{(params_0, m)\})) = true \\ false & else \end{cases}$$

where $|params| = 2$.

The remaining definition of $apply$ corresponds to the informal descriptions given in Appendix C. For all combinations of arguments where there is not definition given, $apply$ yields $null$.

Appendix B ESDL Formal Specification

In this appendix, we give a formal specification of our ESDL. To do so, we introduce a formalization to describe assignments of behavior sequences to ElementOccurrences and Models on the ElementType and Language level, respectively. On this basis, we describe an abstract machine and its state space, and we formally define behaviors and their effects on the state. Finally, we show how an LTS could be derived with our abstract machine.

B.a Assignment of Behavior Sequences to Element Occurrences and Models

We introduce our formalization of assigning behaviors to Element Occurrences and to Models on the Element Type and Language level, respectively. As a foundation, we introduce the concept "scope". Let $Scope$ be a set of [em] Projects.

Let $Behavior$ the set of all possible instances of all Behavior types with all possible arguments. Let $BehaviorSequence$ the set of all sequences of $Behavior$ elements. Let $BehaviorCarrier$ be the union of the set of Languages used in Models in $Scope$'s Projects, and of the set of the ElementTypes in these Languages. Let $BehaviorMapping$ be the set of all possible mappings from $BehaviorCarrier$ to sequences of $Behavior$.

In the following we assume that a user has defined a behavior mapping as some $bm \in BehaviorMapping$.

B.b Introduction into Our Abstract Machine

We introduce some basic definitions required to formally describe the abstract machine's state and its execution semantics.

Let $Subject$ be the union of the set of all Models in the Projects in $Scope$, and of the set of the ElementOccurrences in these Models.

Let $EnablementTask = Subject \times RuntimeInstance \times CustomEnablementData$ where CustomEnablementData is as defined by the user. Let $EnablementTaskList$ be the set of multisets consisting of elements in $EnablementTask$.

Let $EODataMap = \{eo \rightarrow d \mid eo \in ElementOccurrence,\ d \in CustomStorageData\}$, i.e. the set of all possible mappings from ElementOccurrence elements to a CustomStorageData value. Let $DataRepository = \{ri \rightarrow dm \mid ri \in RuntimeInstance,\ dm \in EODataMap\}$, i.e. the set of all possible mappings from RuntimeInstance elements to an $EODataMap$ element.

Let $Event = ElementOccurrence \times String$. Let $EventSequence$ be a sequence of $Event$ elements.

Having the pre-requisites defined, we describe our abstract machine.

Let $SingleState = EnablementTaskList \times DataRepository \times EventSequence$. Then we can define our abstract machine's $StateSpace$ as the powerset of $SingleState$. Given some $stateSet_i \in StateSpace$, the transition of our abstract machine to $stateSet_{i+1} \in StateSpace$ can be defined using a function $trans: StateSpace \to StateSpace$. We establish further concepts to define this function concisely.

We introduce a family of functions $execB_{env}: Behavior \times EnablementTaskList \times DataRepository \times EventSequence \to SateSpace$ with $env \in Environment$. Given a behavior, an initial enablement task list, an initial data repository, and a sequence of previously reported events, each $execB_{env}$ yields a new set of single states. As an intuition, $execB_{env}$ describes the possible effects of executing a single behavior. We define $execB_{env}$ in the following subsections. For brevity, we write $e(b)$ for $execB_{env}(b, tl, dr)$.

We further define a family of functions $execBS_{env}: BehaviorSequence \times EnablementTaskList \times DataRepository \times EventSequence \to SateSpace$ with $env \in Environment$. Given a sequence of behaviors, an initial enablement task list, an initial data repository, and a sequence of previously reported events, each $execBS_{env}$ yields a new set of single states. As an intuition, $execBS_{env}$ describes the effects of executing a sequence of behaviors.

We use a recursive definition for $execBS_{env}$. For an empty behavior sequence, we define $execBS_{env}$ as:

$$execBS_{env}\big((\quad), tl, dr, es\big) = \{(tl, dr, es)\}.$$

For a non-empty behavior sequence, i.e. for $|bs| > 0$, we define $execBS_{env}$ as:

$$execBS_{env}\big((bs_0, \dots, bs_n), tl, dr, es\big) = \bigcup_{\substack{(tl^*, dr^*, es^*) \in \\ execB_{env}(bs_0, tl, dr, es)}} execBS_{env}\big((bs_1, \dots, bs_n), tl^*, dr^*, es^*\big).$$

Let $baseEnv(s, i, d)$ be the environment with the mappings

`CurrentRuntimeInstance` $\to i$,

`EnablementData` $\to d$,

`CurrentModel` $\to s$ if $s \in Model$,

`CurrentObjectOccurrence` $\rightarrow s$ if $s \in ObjectOccurence$,

`CurrentRelationshipOccurrence` $\rightarrow s$ if $s \in RelationshipOccurence$.

Let $t: Subject \rightarrow Language \cup ElementType$ the function where $t(s)$ yields the Language of s if $s \in Model$, or the ElementType of s if $s \in ElementOccurence$. Let $bs(s) = (bs_0, bs_1, \dots, bs_n) = bm\big(t(s)\big)$, describing the sequence of behaviors assigned to the respective s on the ElementType or Language level.

We define the function $singleStateTrans: SingleState \rightarrow StateSpace$ as:

$$singleStateTrans(tl, dr, es) = \bigcup_{t=(s,i,d) \in tl} execBS_{baseEnv(s,i,d)}(bs(s), tl \setminus t, dr, es).$$

As an intuition, $singleStateTrans$ yields for all task in the task list the set of possible states after executing the behavior sequence that is assigned to the respective task's subject on the ElementType or Language level.

Using these building blocks, we can define our abstract machine's transition function now:

$$trans(states_i) = \bigcup_{(tl,dr,es) \in states_i} singleStateTrans(tl, dr, es).$$

As an intuition, $trans$ yields the result of applying $singleStateTrans$ on all states.

B.c Behavior Types

We formally give the semantics of the behaviors by partially defining $execB_{env}$ for each behavior type.

Behavior Type "Enable Element Occurrence". Let $eeo(ri, eo, d, now)$ be a behavior of type "Enable Element Occurrence". Let ri be its "Runtime Instance" argument, eo its "Element Occurrence" argument, d its "Data to pass on" argument, now its "Perform now instead of scheduling it" argument. Then:

$$e\big(eeo(ri, eo, d, true)\big) = execBS_{baseEnv(s^*,i^*,d^*)}(bs(s^*), tl, dr, es),$$

$$e\big(eeo(ri, eo, d, false)\big) = \{(tl \cup \{(s^*, i^*, d^*)\}, dr, es)\}$$

with $s^* = eval(eo, env)$, $i^* = eval(ri, env)$, and $d^* = eval(d, env)$.

Behavior Type "Enable Model". Let $em(new, ri, m, d, now)$ be a behavior of type "Enable Model". Let new be its "Create new runtime instance" argument, ri be its "Runtime Instance"

argument, m its "Model" argument, d its "Data to pass on" argument, now its "Perform now instead of scheduling it" argument. Further let

$$i^* = \begin{cases} eval(ri, env) & \text{if } new = false \\ firstFreeRuntimeInstance(dr) & \text{else} \end{cases},$$

$$dr^* = \begin{cases} dr & \text{if } new = false \\ dr \cup \{firstFreeRuntimeInstance(dr) \rightarrow \{\quad\}\} & \text{else} \end{cases}$$

where $firstFreeRuntimeInstance: DataRepository \rightarrow RuntimeInstance$ is the function that yields $ri_i \in RuntimeInstance$ with i being the number of RuntimeInstances used in the given DataRepository.

Then:

$$e\big(em(new, ri, m, d, true)\big) = execBS_{baseEnv(s^*, i^*, d^*)}(bs(s^*), tl, dr^*, es),$$

$$e\big(em(new, ri, m, d, false)\big) = \{(tl \cup \{(s^*, i^*, d^*)\}, dr^*, es)\}$$

with $s^* = eval(eo, env)$, and $d^* = eval(d, env)$.

Behavior Type "For one item in a collection". Let $fo(vn, c, cb)$ be a behavior of type "For one item in a collection". Let vn be its "Item Variable Name" argument, c be its "Collection" argument, cb its "Child Behaviors" argument. Then:

$$e\big(fo(vn, c, cb)\big) = \bigcup_{cm \in membersOf(eval(c, env))} execBS_{env^*}(cb, tl, dr, es)$$

with $env^* = (env, \{vn \rightarrow cm\})$.

Behavior Type "For each item in a collection". Let $fe(vn, c, cb)$ be a behavior of type "For each item in a collection". Let vn be its "Item Variable Name" argument, c be its "Collection" argument, cb its "Child Behaviors" argument. To concisely describe the "looping" through the collection, we introduce a family of non-deterministic helper functions $feHelper_{env,T}: EnablementTaskList \times DataRepository \times Collection\langle T\rangle \times Identifier \times BehaviorSequence \rightarrow StateSpace$ with $env \in Environment$ and T being some type.

We use a recursive definition for $feHelper_{env,T}$. For an empty collection value, we define $feHelper_{env,T}$ as:

$$feHelper_{env,T}(tl, dr, es, (\quad), vn, cb) = \{(tl, dr, es)\}.$$

For a non-empty behavior sequence, i.e. for $|cv| > 0$, we define $execBS_{env}$ as:

$$feHelper_{env,T}(tl, dr, es, cv, vn, cb) = \bigcup_{\substack{(tl^*, dr^*, es^*) \in \\ execBS_{env^*}(cb, tl, dr)}} feHelper_{env,T}(tl^*, dr^*, es^*, (cv_1, \dots, cv_n), vn, cb)$$

with $env^* = (env, \{vn \to cm\})$. Then we can provide the partial definition for $execB$ that is relevant for behaviors of type "For each item in a collection" as:

$$e\big(fe(vn, c, cb)\big) = feHelper_{env,T}(tl, dr, es, eval(c, env), vn, cb)$$

where T is the type of members of the collection as given by $eval(c, env)$.

Behavior Type "If/Then/Else". Let $if(c, tb, eb)$ be a behavior of type "If/Then/Else". Let c be its "Condition" argument, tb be its "Then Behaviors" argument, eb its "Else Behaviors" argument. Then:

$$e\big(if(c, tb, eb)\big) = \begin{cases} execBS_{env}(tb, tl, dr, es) & \text{if } eval(c, env) = true \\ execBS_{env}(eb, tl, dr, es) & \text{if } eval(c, env) = false \end{cases}$$

Behavior Type "Load Data". Let $ld(ri, eo, vn, cb)$ be a behavior of type "Load Data". Let ri be its "Runtime Instance" argument, eo its "Element Occurrence" argument, vn its "Variable Name" argument, cb its "Child Behaviors" argument.

To concisely describe the approach of looking up data stored for a RuntimeInstance and an ElementOccurrence, and of alternatively creating a new CustomDataInstance if no data was previously stored, we introduce some helper definitions.

We define the helper function $lookupEODataMap$ that maps a RuntimeInstance and a $DataRepository$ element to a $EODataMap$ element:

$$lookupEODataMap(ri, dr) = \begin{cases} dr(ri) & \text{if } \exists x\colon (ri \to x) \in dr \\ \{\ \} & \text{else} \end{cases}$$

We define the helper function $loadOrNew\colon ElementOccurrence \times RuntimeInstance \times DataRepository \to CustomStorageData$ as:

$$loadOrNew(eo, ri, dr) = \begin{cases} dm(eo) & \text{if } \exists x\colon (eo \to x) \in dm \\ Default(CustomStorageData) & \text{else} \end{cases}$$

with $dm = lokupEODataMap(ri, dr)$. Now we can provide the partial definition for $execB$ that is relevant for behaviors of type "Load Data" as:

$$e\big(ld(ri, eo, vn, cb)\big) = execBS_{env^*}(cb, tl, dr, es)$$

with $env^* = (env, \{vn \to loadOrNew(eo, ri, dr)\})$.

Behavior Type "Release Runtime Instance". Let $rri(ri)$ be a behavior of type "Release Runtime Instance". Let ri be its "Runtime Instance" argument.

Let $ris = \{eval(ri, env) \rightarrow x \mid x \in RuntimeInstanceDataEntry\}$, i.e. the set of all possible mappings from the RuntimeInstance as specified by the formula in the argument to some RuntimeInstanceDataEntry. Then:

$$e\big(rri(ri)\big) = \{(tl, dr \setminus ris, es)\}.$$

Behavior Type "Report Event". Let $re(eo, ec)$ be a behavior of type "Report Event". Let eo be its "Element Occurrence" argument, ec its "Event Content" argument. Then:

$$e\big(re(eo, ec)\big) = \left\{\left(tl, dr, \left(es_0, \dots, es_{|es|-1}, e^*\right)\right)\right\}$$

with $e^* = \big(eval(eo, env),\ eval(ec, env)\big)$.

Behavior Type "Store Data". Let $sd(ri, eo, d)$ be a behavior of type "Store Data". Let ri be its "Runtime Instance" argument, eo its "Element Occurrence" argument, d its "Data to be stored" argument.

To concisely describe the formalism of storing data for a RuntimeInstance and an ElementOccurrence, we introduce some helper definitions.

Let $eov = eval(eo, env)$, i.e. the ElementOccurrence value as specified by the formula in the respective argument. Let $eom = \{eov \rightarrow x \mid x \in CustomStorageData\}$ i.e. the set of all possible mappings from eov to a CustomStorageData value. Let $eom^* = \{eov \rightarrow eval(d, env)\}$, i.e. the set containing a mapping from eov to the new data as specified by the formula d.

Let ris be defined as for the behavior type "Release Runtime Instance". Let $ris^* = (ris \setminus eom) \cup eom^*$, i.e. a set of ElementOccurrence to CustomStorageData mappings where the mapping from eov is replaced with the evaluation of the formula in the "Data to be stored" argument. Then:

$$e\big(sd(ri, eo, d)\big) = \{(tl, (dr \setminus ris) \cup ris^*, es)\}.$$

B.d Deriving an LTS using Our Abstract Machine

Given some initial Model M in a $Scope$'s Project, let $tl_0 = \big\{\big(M, ri_0, Default(CustomEnablementData)\big)\big\}$ be an initial $EnablementTaskList$. Let $dr_0 = \{ri_0 \rightarrow \{\ \}\}$ be an initial $DataRepository$. Let $es_0 = (\ )$ be an initial empty EventSequence. Then let $stateSet_0 = \{(tl_0, dr_0, es_0)\}$ be the initial $StateSpace$.

The LTS could be specified based on the sequences of reported events es of $(tl, dr, es) \in stateSet_i$ derived from transitioning the abstract machine starting with $stateSet_0$. This however is just a theoretical way of specifying the LTS that cannot directly be implemented in software. For example, if the given behaviors entail a loop, the resulting sequences of events will become infinite. Because this will result in the LTS becoming infinite as well, a computer could not store a direct representation of the LTS. An actual model checker must therefore derive a finite LTS from the given behaviors to allow solving typical model checking problems. For performance reasons, an actual model checker should also detect if branches in the tree merge again at some point to reduce the number of states that need to be processed. The CADP model checker used by our implementation takes care of these aspects.

Also, in our definitions for the abstract machine, there are some gaps that were deliberately left undefined for conciseness. If an implementation of our abstract machine runs into such a gap, it must stop its operation and should issue a message giving information about the situation which led to running into the respective gap.

For example, if an implementation of our abstract machine is requested to enable a wrongly specified ElementOccurrence, then it is unclear what it should do. In such a case, stopping its operation makes sense. This example can be formally described with an “Element Occurrence” argument of an “Enable Element Occurrence” behavior evaluating to $null$. In this case, s^* would be $null$. Because $bs(s^*)$ with $s^* = null$ is not defined, the abstract machine implementation would run into a definition gap and would have to stop its operation as per our specification. In our implementation, we handle such gaps accordingly.

Appendix C Reference on Data Types in Our Languages

In this appendix, we provide tabular references for properties, functions and lambdas supported by the data types of our languages.

Let a "predecessor expression" be an expression that evaluates to an instance of some specific data type. We call this data type's "properties" all the supported property accessors that can be appended to the type's predecessor expressions. Similarly, we call this data type's "functions" and "lambdas" all the supported function and lambda accessors that can be appended to the type's predecessor expressions, respectively. We denote with "predecessor value" the value that a predecessor expression evaluates to.

C.a Boolean

C.a.a Properties

Name	Return Type	Description
Inverse	Boolean	Returns the inverse of the predecessor value.
AsString	String	Returns the String representation of the predecessor value, i.e. either `true` or `false`.

C.a.b Functions

Name	Parameters with Types	Return Type	Description
And	other: Boolean	Boolean	Returns $null$ if $other$ is $null$. Else, returns the result of logical conjunction of the predecessor value and $other$.
Or	other: Boolean	Boolean	Returns $null$ if $other$ is $null$. Else, returns the result of the logical disjunction of the predecessor value and $other$.
Xor	other: Boolean	Boolean	Returns $null$ if $other$ is $null$. Else, returns the result of applying exclusive or to the predecessor value and $other$.
Equals	other: Boolean	Boolean	Returns $null$ if $other$ is $null$. Returns $true$ if the predecessor value and $other$ are equal. Else, returns $false$.
IfElse	ifTerm: <T>, elseTerm: <T>	<T>	Returns $ifTerm$ if the predecessor value is $true$. Returns $elseTerm$ if the predecessor value is $false$. <T> is a placeholder for a single arbitrary type.

C.b Integer

C.b.a Properties

Name	Return Type	Description
Negation	Integer	Returns the negation of the predecessor value.
AsString	String	Returns a String representation of the predecessor value as provided by the LNT-internal transformation according to (Champelovier et al. 2017).

C.b.b Functions

Name	Parameters with Types	Return Type	Description
Equals	other: Integer	Boolean	Returns *null* if *other* is *null*. Returns *true* if the predecessor value and *other* are equal. Else, returns *false*.
Unequals	other: Integer	Boolean	Returns *null* if *other* is *null*. Returns *false* if the predecessor value and *other* are equal. Else, returns *true*.
GreaterThan	other: Integer	Boolean	Returns *null* if *other* is *null*. Else, returns the truth value of the predecessor value being greater than *other*.
LessThan	other: Integer	Boolean	Returns *null* if *other* is *null*. Else, returns the truth value of the predecessor value being less than *other*.
GreaterThanOrEquals	other: Integer	Boolean	Returns *null* if *other* is *null*. Else, returns the truth value of the predecessor value being greater than or equals *other*.
LessThanOrEquals	other: Integer	Boolean	Returns *null* if *other* is *null*. Else, returns the truth value of the predecessor value being less than or equals *other*.
Plus	other: Integer	Integer	Returns *null* if *other* is *null*. Else, returns the result of adding *other* to the predecessor value.
Minus	other: Integer	Integer	Returns *null* if *other* is *null*. Else, returns the result of subtracting *other* from the predecessor value.
Times	other: Integer	Integer	Returns *null* if *other* is *null*. Else, returns the result of multiplying the predecessor value with *other*.
DividedBy	other: Integer	Integer	Returns *null* if *other* is *null*. Else, returns the result of dividing the predecessor value by *other*.
Modulo	other: Integer	Integer	Returns *null* if *other* is *null*. Else, returns the result of applying the modulo operator to the predecessor value and *other*.

C.c Double

C.c.a Properties

Name	Return Type	Description
Negation	Double	Returns the negation of the predecessor value.
AsString	String	Returns a String representation of the predecessor value as provided by the LNT-internal transformation according to (Champelovier et al. 2017).

C.c.b Functions

Name	Parameters with Types	Return Type	Description
Equals	other: Double	Boolean	Returns *null* if *other* is *null*. Returns *true* if the predecessor value and *other* are equal. Else, returns *false*.
Unequals	other: Double	Boolean	Returns *null* if *other* is *null*. Returns *false* if the predecessor value and *other* are equal. Else, returns *true*.
GreaterThan	other: Double	Boolean	Returns *null* if *other* is *null*. Else, returns the truth value of the predecessor value being greater than *other*.
LessThan	other: Double	Boolean	Returns *null* if *other* is *null*. Else, returns the truth value of the predecessor value being less than *other*.
GreaterThanOrEquals	other: Double	Boolean	Returns *null* if *other* is *null*. Else, returns the truth value of the predecessor value being greater than or equals *other*.
LessThanOrEquals	other: Double	Boolean	Returns *null* if *other* is *null*. Else, returns the truth value of the predecessor value being less than or equals *other*.
Plus	other: Double	Double	Returns *null* if *other* is *null*. Else, returns the result of adding *other* to the predecessor value.
Minus	other: Double	Double	Returns *null* if *other* is *null*. Else, returns the result of subtracting *other* from the predecessor value.
Times	other: Double	Double	Returns *null* if *other* is *null*. Else, returns the result of multiplying the predecessor value with *other*.
DividedBy	other: Double	Double	Returns *null* if *other* is *null*. Else, returns the result of dividing the predecessor value by *other*.

C.d String

C.d.a Properties

Name	Return Type	Description
Length	Integer	Returns the length of the String as determined with the LNT-internal `length` function according to (Champelovier et al. 2017).

C.d.b Functions

Name	Parameters with Types	Return Type	Description
Equals	other: String	Boolean	Returns *null* if *other* is *null*. Returns *true* if the predecessor value and *other* are equal. Else, returns *false*.
Unequals	other: String	Boolean	Returns *null* if *other* is *null*. Returns *false* if the predecessor value and *other* are equal. Else, returns *true*.

Name	Parameters with Types	Return Type	Description
ConcatenatedWith	other: String	String	Returns *null* if *other* is *null*. Else, returns the concatenation of the predecessor value and *other*.
Substring	startIndex: Index	String	Returns *null* if *startIndex* is *null* or if *startIndex* < 0 or if *startIndex* > length of the predecessor value. Else, returns the substring of the predecessor value that starts at index *startIndex*.
Substring	startIndex: Integer, length: Integer	String	Returns *null* if *startIndex* is *null* or if *length* is *null* or if *startIndex* < 0 or if *length* < 0 or if *startIndex* > length of the predecessor value or if *startIndex* + *length* > length of the predecessor value. Else, returns the substring of the predecessor value that starts at index *startIndex* and has the length *length*.

C.e Collection<T>

T is a placeholder for a single arbitrary type.

C.e.a Properties

Name	Return Type	Description
Count	Integer	Returns the number of members in the predecessor value.
Head	T	Returns *null* if the predecessor value is empty. Else, returns the first member of the predecessor value.
Tail	Collection<T>	Returns *null* if the predecessor value is empty. Else, returns the predecessor value without its first member.

C.e.b Functions

Name	Parameters with Types	Return Type	Description
AppendedWith	other: Collection<T>	Collection<T>	Returns *null* if *other* is *null*. Else, returns predecessor value appended with *other* using the LNT `union` function according to (Champelovier et al. 2017).

C.e.c Lambdas

Each of the lambdas iterate through the predecessor value's members and evaluates the inner formula in such a way that the given parameter maps to the respective member.

Name	Parameters with Types	Expected Lambda Body Type	Return Type	Description
All	item: T	Boolean	Boolean	Returns *null* if the inner formula evaluates to *null* during iteration. Returns *false* if the inner formula evaluates to *false* during iteration. Else, returns *true*.
Any	item: T	Boolean	Boolean	Returns *null* if the inner formula evaluates to *null* during iteration. Returns *true* if the inner formula evaluates to *true* during iteration. Else, returns *false*.
Single	item: T	Boolean	T	Returns *null* if the inner formula evaluates to *null* during iteration. Returns the first iteration's member that the inner formula evaluates to *true* for. Else, returns *null*.
Where	item: T	Boolean	Collection<T>	Returns *null* if the inner formula evaluates to *null* during iteration. Else, returns the predecessor value that is filtered so that only those members remain that the inner formula evaluates to *true* for.
Select	item: T	U	Collection<U>	Returns *null* if U is a Collection type. Else, returns a collection where each member of the predecessor value is mapped to the evaluation result of the inner formula. U is a placeholder for a single arbitrary type. In the current implementation, specifying a collection type for U is not supported, i.e. you cannot describe collections of collections with Select.

C.f [em] Data Types

The [em] Data Types in our languages closely reflect the data model of [em] as introduced in section 4.1. Most of its aspects can be described in a templatic fashion.

We give templates that describe the main properties of each type in the first sub-subsection. We describe additional properties and functions in the remaining sub-subsections. We use bracket-enclosed text for placeholders.

C.f.a Templates for Main Properties

Name	Existence Criteria	Return Type	Description
[Association Name]	For each association with max. target multiplicity of one	[Association's target class]	Returns the association's target instance, or *null* if the association does not have a target instance.
[Association Name]	For each association with max. target multiplicity greater than one	Collection<[Association's target class]>	Returns a collection of the association's target instances.
[Field Name]	For each field (also of some parent class)	String	Returns the field's value.

C.f.b Additional Element Properties

Name	Return Type	Description
IsRelationship	Bool	Returns $true$ if the Element is a Relationship; $false$ otherwise.
IsObject	Bool	Returns $true$ if the Element is an Object; $false$ otherwise.
AsRelationship	Relationship	Returns a Relationship corresponding to this Element if it is a Relationship; $null$ otherwise.
AsObject	Relationship	Returns an Object corresponding to this Element if it is an Object; $null$ otherwise.

C.f.c Additional ElementOccurrence Properties

Name	Return Type	Description
IsRelationshipOccurrence	Bool	Returns $true$ if the predecessor value is a RelationshipOccurrence; $false$ otherwise.
IsObjectOccurrence	Bool	Returns $true$ if the predecessor value is an ObjectOccurrence; $false$ otherwise.
AsRelationshipOccurrence	Relationship	Returns a RelationshipOccurrence corresponding to the predecessor value if it is a RelationshipOccurrence; $null$ otherwise.
AsObjectOccurrence	Relationship	Returns an ObjectOccurrence corresponding to the predecessor value if it is a ObjectOccurrence; $null$ otherwise.

C.f.d Additional ElementType Properties

Name	Return Type	Description
IsRelationshipType	Bool	Returns $true$ if the predecessor value is a RelationshipType; $false$ otherwise.
IsObjectType	Bool	Returns $true$ if the predecessor value is an ObjectType; false otherwise.
AsRelationshipType	Relationship	Returns a RelationshipType corresponding to the predecessor value if it is a RelationshipType; $null$ otherwise.
AsObjectType	Relationship	Returns an ObjectType corresponding to the predecessor value if it is a ObjectType; $null$ otherwise.

C.f.e Additional ElementType, ObjectType and RelationshipType Functions

Name	Parameters with Types	Return Type	Description
Equals	other: [Type of predecessor value]	Bool	Returns $null$ if $other$ is $null$. Returns $true$ if the predecessor value and $other$ are equal. Else, returns $false$.

C.g Runtime-relevant Types

RuntimeInstances do not have any properties, functions, or lambdas. Custom types have for each of their field a property and a function. We introduce them in the following two sub-subsections.

C.g.a Custom Type Properties

Name	Existence Criteria	Return Type	Description
[Field Name]	For each field	[Field type]	Returns the value of the respective field.

C.g.b Custom Type Functions

Name	Existence Criteria	Parameters with Types	Return Type	Description
WithChanged_[Field Name]	For each field	value: [Field type]	[Type of predecessor value, i.e. either CustomEnablementData, or CustomStorageData]	Returns a copy of the custom type instance where the respective field value is replaced with the argument $value$.

Appendix D Source Code of Plugin

The source code of the plugin should come with this document in digital form.

If it is missing, please contact Hauke Pribnow.

Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this Master Thesis titled "Leveraging Propositional Logic-Based Model Checking to Enable Convenient Analysis of Process Models in Arbitrary Graph-Based Process Modeling Languages" is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references.

Münster, 2017-12-19

Hauke Pribnow

Consent Form

for the use of plagiarism detection software to check my thesis

Name: Pribnow **Given Name:** Hauke

Student number: 365292 **Course of Study:** Information Systems

Address: Averkampstraße 9, 48151 Münster, Germany

Title of the thesis: Leveraging Propositional Logic-Based Model Checking to Enable Convenient Analysis of Process Models in Arbitrary Graph-Based Process Modeling Languages

What is plagiarism? Plagiarism is defined as submitting someone else's work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software. The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions. Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as "failed". This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 2017-12-19

Hauke Pribnow