

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

A Domain-Specific Language for Business Process Model and Notation

vorgelegt von

Marlon Kranz

MIN-Fakultät

Fachbereich Informatik

Studiengang: Wirtschaftsinformatik

Abgabedatum: 27.10.2026

Erstgutachter: Dr. Oliver Kopp

Zweitgutachter: Christian Rahe

Abstract

Business Process Model and Notation (BPMN) 2.0 models are drawn as diagrams, exchanged as XML, and executed by BPMN engines. This thesis asks how developers who know control flow from programming languages, but have not learned BPMN, can write executable BPMN 2.0 processes in a textual notation. Such a notation has to meet three requirements: control flow written as statements, XML that an engine runs as is, and an import of existing XML into editable source. None of the existing textual notations we found satisfies all three requirements. We present BPMNscript, a textual domain-specific language (DSL) for the Operaton engine. Developers write statements such as `if`, `while`, and `parallel`, from which a compiler derives gateways and sequence flows. A language server provides editor support in Visual Studio Code. The DSL covers 17 of the 19 element types that Operaton executes, and the compiled XML runs on Operaton without any additional libraries or plugins of the DSL. Existing models are imported into source. This is the difficult direction, since BPMN allows almost arbitrary graphs and not every graph has a block-structured equivalent. We judge each round trip from XML through the DSL back to XML with a fixed equivalence relation, and deploy each rebuilt mined and official model on Operaton 2.1.0. Of the 6,060 mined models from GitHub that Operaton deploys, the importer accepts 4,946, and 3,858 of these count as equivalent, about four in five. For 3,803 of the equivalent models, the rebuilt model gains a history time to live, the retention period for the records of finished instances, which Operaton requires by default. Of the 921 divergent models, 254 differ only in content without effect on execution. The share of equivalent models falls with size, from 77.3 % of the deployed models with up to 10 flow nodes to 5.9 % above 50. All 1,996 valid generated programs count as equivalent, and at most four diverge when the order of their elements changes. Of the 2,151 accepted official models of Operaton and Camunda, 93.7 % count as equivalent. Developers can thus write processes for Operaton as text and import most small existing models. Most models above 50 flow nodes are refused or diverge, which marks where the scope and the recovery of statements from gateway graphs have to grow.

Contents

Abbreviations **xiii**

1 Introduction **1**
1.1 Problem and Requirements . . . 2
1.2 BPMNscript in Brief . . . 3
1.3 Research Questions . . . 6
1.4 Research Method . . . 7
1.5 Contributions . . . 8
1.6 Summary of Results . . . 9
1.7 Thesis Outline . . . 10

2 Background **11**
2.1 BPMN 2.0: Notation, Interchange Format, Execution Semantics . . . 11
2.2 Text and Diagrams for Readers . . . 13
2.3 Block-Structured and Graph-Based Process Languages . . . 14
2.4 Engine Conformance Gap . . . 16
2.5 The Operaton BPMN Engine . . . 17
2.6 Language Workbenches . . . 18
2.7 The Workflow-as-Code Paradigm . . . 19

3 Related Work **21**
3.1 BPELscript and the BPEL Precedent . . . 21
3.2 StalkCD: CI/CD Pipelines Between Jenkinsfiles and BPMN . . . 22
3.3 Process Languages Outside BPMN . . . 23
3.4 Textual Notations and Code Front Ends for BPMN and Related Notations . . . 25
3.5 Zeebe's YAML Format . . . 29
3.6 BPMN Sketch Miner . . . 29
3.7 kUML . . . 30
3.8 HyLiMo and Renew . . . 30
3.9 LLM-Driven BPMN Generation . . . 31
3.10 Restructuring and Bidirectional Transformation . . . 32
3.11 One Process in Four Notations . . . 33
3.12 Gap Analysis and Positioning . . . 40

4 Approach **41**
4.1 Design Objectives and Non-Goals . . . 41

4.2 Scoping the Executable BPMN Subset . . . 42
4.2.1 The Executable Subset of Operaton . . . 43
4.2.2 Conformance Sub-Classes as a Scope . . . 44
4.2.3 Empirical Secondary Justification . . . 45
4.2.4 Coverage and Exclusions . . . 46
4.3 DSL Syntax Design . . . 47
4.3.1 Running Example . . . 47
4.3.2 Design Rationale . . . 47
4.3.3 Lexical and Grammatical Structure . . . 52
4.3.4 Naming, References, Scoping . . . 53
4.3.5 Operaton-Specific Constructs . . . 56
4.4 The Intermediate Representation . . . 57
4.5 Bidirectional Transformation Strategy . . . 59
4.5.1 DSL to BPMN 2.0 XML . . . 59
4.5.2 BPMN 2.0 XML to DSL . . . 60
4.5.3 Diagram Interchange Handling . . . 64
4.5.4 Equivalence Relation . . . 64

5 Implementation **69**
5.1 Architecture Overview . . . 69
5.2 Language Front End with Langium . . . 70
5.2.1 From Grammar File to Parser . . . 71
5.2.2 Scoping and Cross-Reference Resolution . . . 72
5.2.3 Validation and Diagnostics . . . 72
5.3 IDE Integration via LSP . . . 74
5.3.1 Syntax Highlighting and Semantic Tokens . . . 74
5.3.2 Completion, Navigation, Hover, and Outline . . . 76
5.4 Compiler Back End: BPMN 2.0 XML Generation . . . 77
5.4.1 Lowering the Syntax Tree to the IR . . . 77
5.4.2 IR-to-XML Mapping Rules . . . 78
5.4.3 Operaton Extension Attributes . . . 78
5.4.4 Auto-Layout for Diagram Interchange . . . 79
5.5 Import: BPMN 2.0 XML to DSL . . . 79
5.5.1 Reading: Refusals, Warnings, and the Camunda Namespace . . . 79
5.5.2 Restructuring the Graph into Blocks . . . 80
5.6 Command-Line Tool and Conversion Commands . . . 82
5.7 Use of AI-Assisted Coding . . . 82
5.8 Testing Infrastructure . . . 84

6 Evaluation **85**
6.1 Evaluation Design and Equivalence Criteria . . . 85
6.2 Round Trips of Generated Programs . . . 87
6.3 Round-Trip Correctness on Real-World BPMN . . . 88
6.3.1 Corpus Selection . . . 88

6.3.2 Purposive Sets . . . 91
6.3.3 Methodology and Tolerances . . . 91
6.3.4 Results . . . 92
6.4 BPMN MIWG Interchange Conformance . . . 96
6.5 Notation Size . . . 97
6.6 Discussion . . . 98
6.7 Threats to Validity . . . 101

7 Conclusion and Future Work **103**
7.1 Summary of Contributions . . . 103
7.2 Answers to the Research Questions . . . 103
7.3 Limitations . . . 105
7.4 Future Work . . . 106
7.4.1 AI-Assisted BPMN Authoring with Compiler Feedback . . . 106
7.4.2 Multi-Engine Portability . . . 107
7.4.3 Collaboration, Pools, and Lanes . . . 107
7.4.4 Bidirectional Diagram Interchange Preservation . . . 107
7.4.5 Wider Import Coverage . . . 107
7.4.6 Complete Structuring of Arbitrary Graphs . . . 108
7.4.7 Test Generation, Formatting, and Embedded Script Tooling . . . 108

A Event Combinations **109**

B Element Types **111**

C Compiled Running Example **113**

D Syntax Candidates **117**

E Tolerances of the Equivalence Relation **125**

F Restructuring Rules of the Printer **129**

G Results in All Three Counts **131**

H Architectural Decision Records **133**

I Use of AI Tools **135**

Bibliography **137**

List of Figures

1.1 The three roles of BPMN 2.0, connected through the standardized XML 2
1.2 The running example after compilation 5
1.3 The position of the textual DSL 6

2.1 A small process in the BPMN notation 12
2.2 A structured and an unstructured process model 15

4.1 The delivered tools 42
4.2 Statements of the DSL and the BPMN fragments they compile to 54
4.3 The `if` statement of the running example in the intermediate representation . . 58
4.4 The four transforms of the DSL meet in one intermediate representation 59
4.5 A graph-based variant of the running example, as a modeler might draw it . . . 62

5.1 Packages and their dependencies 70
5.2 Editor support in Visual Studio Code 75

6.1 The steps of a round trip 86
6.2 A dropped loop exit 94

List of Tables

3.1 Five constructs in four notations . . . 37
3.2 Related textual process languages compared . . . 38

4.1 Element types by outcome in Operaton . . . 45
4.2 The six candidate syntaxes . . . 50
4.3 Statement kinds of the DSL . . . 53
4.4 Chosen and rejected forms of DSL constructs . . . 55
4.5 Fields compared by the equivalence relation, per node kind of the IR . . . 65
4.6 The ten tolerances of the equivalence relation . . . 66

5.1 Packages of the implementation . . . 69
5.2 Editor features . . . 76

6.1 The six outcomes of a round trip . . . 86
6.2 Where the equivalence relation departs from execution-relevant content . . . 87
6.3 Round trips of 2,000 generated programs per ordering of the imported model, by outcome . . . 87
6.4 Public collections of BPMN models considered for the evaluation . . . 89
6.5 Status of the 74,988 non-empty distinct contents of the frame at acquisition . . . 90
6.6 Outcomes of the 6,060 corpus models that Operaton deploys, each with its share of the deployed models . . . 92
6.7 Tolerances needed by the corpus models . . . 93
6.8 Outcomes of the deployed corpus models by size band . . . 93
6.9 Causes of the differing conditions in the divergent corpus models . . . 94
6.10 Warnings of the importer by category . . . 95
6.11 Printed source of the accepted corpus models by size band . . . 95
6.12 Outcomes per purposive set . . . 96
6.13 Outcome of each of the 22 MIWG reference models . . . 97

A.1 The ten positions of an event counted in Section 4.2.1 . . . 109
A.2 Outcome of every event combination . . . 110

B.1 Outcome of every element type . . . 112

G.1 Outcomes of the deployed corpus models in the three counts . . . 131
G.2 Tolerances needed by the corpus models in the three counts . . . 132

H.1 The ADRs of BPMNscript at release 0.1.1 133

I.1 Generative AI tools, their use, and where they are used 135

List of Listings

1.1 The running example: an expense claim written in BPMNscript 4

3.1 The loan approval of BPELscript 23
3.2 The loan approval of BPELscript in the DSL 24
3.3 The example workflow of the WEE DSL 26
3.4 The example workflow of the WEE DSL in the DSL 27
3.5 The invoice approval in kUML 31
3.6 The invoice approval in the DSL 34
3.7 The invoice approval in the Workflow DSL 35
3.8 The invoice approval with the fluent builder 36
3.9 The invoice approval in Loadshift 36

4.1 Excerpt of the `order-handling` example published with BPMNscript, in the chosen syntax 48
4.2 A user task, a sub-process with one child, and an error handler in variant B, the chosen syntax 49
4.3 A user task, a sub-process with one child, and an error handler in variant D, the off-side rule 49
4.4 A user task, a sub-process with one child, and an error handler in variant F, the binding form 50
4.5 The elements generated for `if (amount > 500)` of Listing 1.1 60
4.6 The source printed for Figure 4.5, with one blank line added 62
4.7 The printed source for a parallel fork with an exclusive merge 63

5.1 Three rules of the grammar file, with comments omitted and one long line broken 71
5.2 A faulty source and its diagnostics 73
5.3 The validator's message for a second plain or timer start in a process 73
5.4 The import's refusal of an exclusive split 80

D.1 Variant A, colon configuration: the `order-handling` process (71 lines). 117
D.2 Variant B, parentheses for attributes, braces for children: the `order-handling` process (53 lines). 118
D.3 Variant C, keyword prose: the `order-handling` process (53 lines). 119
D.4 Variant D, off-side rule (indentation): the `order-handling` process (79 lines). . . 120
D.5 Variant E, decorator annotations: the `order-handling` process (88 lines). 122
D.6 Variant F, binding form: the `order-handling` process (51 lines). 124

Abbreviations

Notation	Description
ADR	architectural decision record
AI	artificial intelligence
ANTLR	ANother Tool for Language Recognition
API	application programming interface
AST	abstract syntax tree
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CI/CD	continuous integration and continuous delivery
CMMN	Case Management Model and Notation
CPEE	Cloud Process Execution Engine
DI	Diagram Interchange
DSL	domain-specific language
FIM	Federal Information Management
IDE	integrated development environment
IR	intermediate representation
JSON	JavaScript Object Notation
JUEL	Java Unified Expression Language
JVM	Java Virtual Machine
LLM	large language model
LSP	Language Server Protocol
MIWG	Model Interchange Working Group
OMG	Object Management Group
REST	Representational State Transfer

Notation	Description
RPST	Refined Process Structure Tree
SHA-1	Secure Hash Algorithm 1
TOSCA	Topology and Orchestration Specification for Cloud Applications
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WEE	Workflow Execution Engine
WSDL	Web Services Description Language
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

1 Introduction

Business Process Model and Notation (BPMN) is a widely accepted notation for the graphical *modeling* of business processes in industry and academia [GHLW18, p. 250]. It is a graphical notation [OMG13, Sect. 7.1], meant to be "readily understandable by all business users" [OMG13, Sect. 1.1]. Among these users, the specification lists the business analysts who draft processes, the technical developers who implement them, and the business people who manage and monitor them [OMG13, Sect. 1.1]. A BPMN diagram is built from activities, events, and gateways, connected by sequence flows and grouped by pools and lanes [OMG13, Sect. 7.3]. Its models serve two audiences: conceptual models help business stakeholders understand and communicate a process, whereas executable models are produced by technical stakeholders such as software developers for deployment on a process engine [DLMR18, p. 79].

BPMN 2.0 plays three roles in the life of a process model: modeling, interchange, and orchestration (Figure 1.1). Modeling is the first role: drawing the process in the graphical notation described above. Beyond the diagram, the BPMN 2.0 specification defines a standardized Extensible Markup Language (XML) *interchange format* that serializes the model and its Diagram Interchange (DI) information into a vendor-neutral representation [OMG13, Sects. 2.2.8, 15]. This second role, BPMN as an interchange format, decouples the modeling tool from any later processing step: a model drawn in one tool can be opened in another, checked by static analysis tools such as BPMNspector [GNV17], or handed to an engine. The Object Management Group (OMG) founded the BPMN Model Interchange Working Group (MIWG) [OMG26] to make modeling tools interoperable, and the group tests whether tools read, write, and reproduce a set of reference models in a serialization that conforms to the standard [GNV17, p. 5]. Geiger et al. [GHLW18, p. 252] state that BPMN is standardized to make models portable and tools interoperable across modeling tools and engines, and a tool vendor argues on this basis that following the standard prevents lock-in to one modeling tool [Dee24].

Since BPMN version 2.0, the same XML representation also serves a third role: it is directly executable by a BPMN engine that *orchestrates* the modeled activities at runtime [Har17, pp. 4, 25; GHLW18, p. 250]. Engines such as the open-source BPMN engine Operaton [Ope25] consume BPMN 2.0 XML, interpret its execution semantics, and drive the orchestration of services and tasks. The standardized XML ties those three roles together. Interchange and orchestration are defined only in terms of this XML, so a tool can change how a model is authored without changing how it is exchanged or executed (Figure 1.1). This thesis takes that route: it changes the modeling role and leaves interchange and orchestration to the standardized XML.

For its modeling role, BPMN presents a process to its users as a diagram [OMG13, Sect. 7.1]. Whether a diagram or a text serves a reader better depends on the task [ISP22, pp. 1895, 1897]

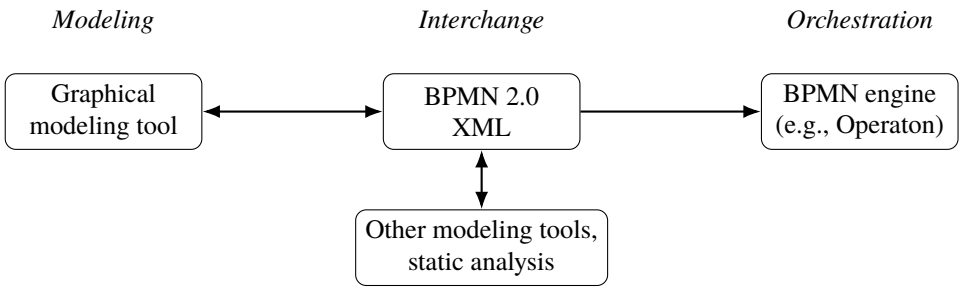


Figure 1.1: The three roles of BPMN 2.0, connected through the standardized XML.

and on the reader (Section 2.2). Petre [Pet95, p. 42] concludes that reading a graphical notation is an acquired skill. In an experiment by Ottensooser et al. [OFR+12, pp. 596, 603], only readers trained in process modeling understood a process significantly better after reading its BPMN diagram, whereas both trained and untrained readers gained from written use cases. This thesis targets developers who know control flow from programming languages but have not learned the BPMN notation. Such developers are among the technical stakeholders who produce executable models [DLMR18, p. 79]. We assume that a notation built from the control-flow statements of programming languages suits them better than the diagram. Dumas et al. [DLMR18, p. 18] object that textual process descriptions are easy to misinterpret because free-form text is ambiguous. In a formal language, every statement maps to a defined BPMN construct, so this ambiguity does not arise. For developers, textual models also show more detail per screen and merge line by line in version control [GKR+07, Sect. 2].

1.1 Problem and Requirements

BPMN does not define a textual notation for authoring: the specification only defines its XML serialization as an interchange format to exchange models between tools [OMG13, Sects. 2.2.8, 15]. Developers who know control flow from programming languages and prefer working in text therefore have no notation in which they write the control flow of an executable BPMN process as statements and still exchange it with modeling tools and run it on an engine. Among the decision patterns for developing a domain-specific language (DSL), Mernik et al. [MHS05, p. 323] include turning an existing visual notation into a textual one. A textual notation for BPMN that serves developers has to meet the following requirements.

R1: Control flow is written with statements in the style of programming languages, and the gateways and sequence flows of the model follow from them.

R2: The notation compiles to BPMN 2.0 XML that an existing engine executes as is, without a library or plugin of the notation at runtime.

R3: Executable BPMN 2.0 XML can be imported, that is, translated back into editable source.

R1 rests on the assumption above that statements from programming languages suit these developers better than the diagram. Under R1, statements such as `if` and `while`, and `parallel` blocks, express the control flow, so the author writes no gateway and no sequence flow except an explicit `goto`, a jump to a statement by its name. Bischof et al. [BKLL09, p. 42] note that "representing graphs in a textual syntax is equivalent to programming using goto statements," so R1 limits explicit edges to these jumps. R2 protects the other two roles: interchange and orchestration both rest on the standardized XML, so a notation that compiles to it changes neither. A notation that needs its own library or plugin at runtime ties orchestration to that library, so the engine cannot run the XML on its own. R3 is needed because many models already exist and because processes are often drawn in a modeling tool before a developer adds the execution details. BPMN models exist in large numbers: Türker et al. [TVH22b, p. 68] found 79,713 distinct BPMN models in 18,534 open-source repositories on GitHub, and without import, a textual notation could serve only processes written in it from scratch. In practice, a process is often drawn in a BPMN modeling tool before the details the engine needs are added [GHLW18, p. 252], so a textual notation for the developer who adds these details has to start from the diagram. R1 is also what makes R3 hard, since a structured syntax has to recover statements from a gateway graph that may have no structured equivalent [KHB00, Theorem 2; PGD12, p. 521, Fig. 5].

No textual notation we found meets all three requirements, and most meet one or none of them (Chapter 3 and Section 3.12).

1.2 BPMNscript in Brief

This thesis presents BPMNscript, a textual DSL for executable BPMN 2.0 processes, together with a bidirectional transformation between the DSL and BPMN 2.0 XML. The compiler derives the gateways and sequence flows of a process from its structured statements and adds a generated diagram layout. The importer reads existing BPMN 2.0 XML, recovers structured statements from its gateway graph, and writes a `goto` where no structured statement fits. Every statement inherits its semantics from the BPMN 2.0 XML it compiles to: the DSL replaces the notation in which a process is written but retains the execution semantics of BPMN.

Listing 1.1 shows an expense claim written in the DSL. From it, the compiler generates the process in Figure 1.2. The source contains no gateway and no sequence flow, and the compiler derives them from the order and nesting of the statements. The DSL is thus a second notation for the modeling role, and the XML it generates remains the format for interchange with modeling tools and for orchestration on an engine (Figure 1.3).

BPMNscript targets one engine, Operaton [Ope25], an open-source BPMN 2.0 engine [Ope26a] that continues the code base of Camunda 7 [Ope25] (Section 2.5). A single engine is targeted because engines execute BPMN differently. The specification describes its execution semantics informally, in prose [OMG13, Sect. 13.1], so engines that implement it cannot be expected to behave identically [GHLW18, p. 251]. Of 70 features tested on three open-source engines, only 30 are supported by all three [GHL+15, pp. 24, 28] (Section 2.4). Operaton *deploys* a BPMN 2.0 XML

List. 1.1: The running example: an expense claim written in BPMNscript [Kra26b].

```
process expense-claim {
  var amount: number
  var compliant: boolean

  start ClaimSubmitted {
    form {
      amount: number "Claimed amount"
    }
  }

  service CheckPolicy(label: "Check the expense policy", class:
  ↪ "com.example.expense.CheckPolicyDelegate")

  while (!compliant) {
    user ReviseClaim(label: "Revise the claim", assignee: "demo") {
      form {
        amount: number "Corrected amount"
        compliant: boolean "Claim now complies"
      }
    }
  }

  if (amount > 500) {
    user ApproveClaim(label: "Approve the claim", assignee: "manager")
  } else {
    service AutoApprove(label: "Approve automatically", class:
    ↪ "com.example.expense.AutoApproveDelegate")
  }

  parallel {
    {
      service BookPayment(label: "Book the payment", class: "com.example.expense.BookPaymentDelegate")
    }
    {
      service NotifyClaimant(label: "Notify the claimant", class:
      ↪ "com.example.expense.NotifyDelegate")
    }
  }

  end ClaimSettled
}
```

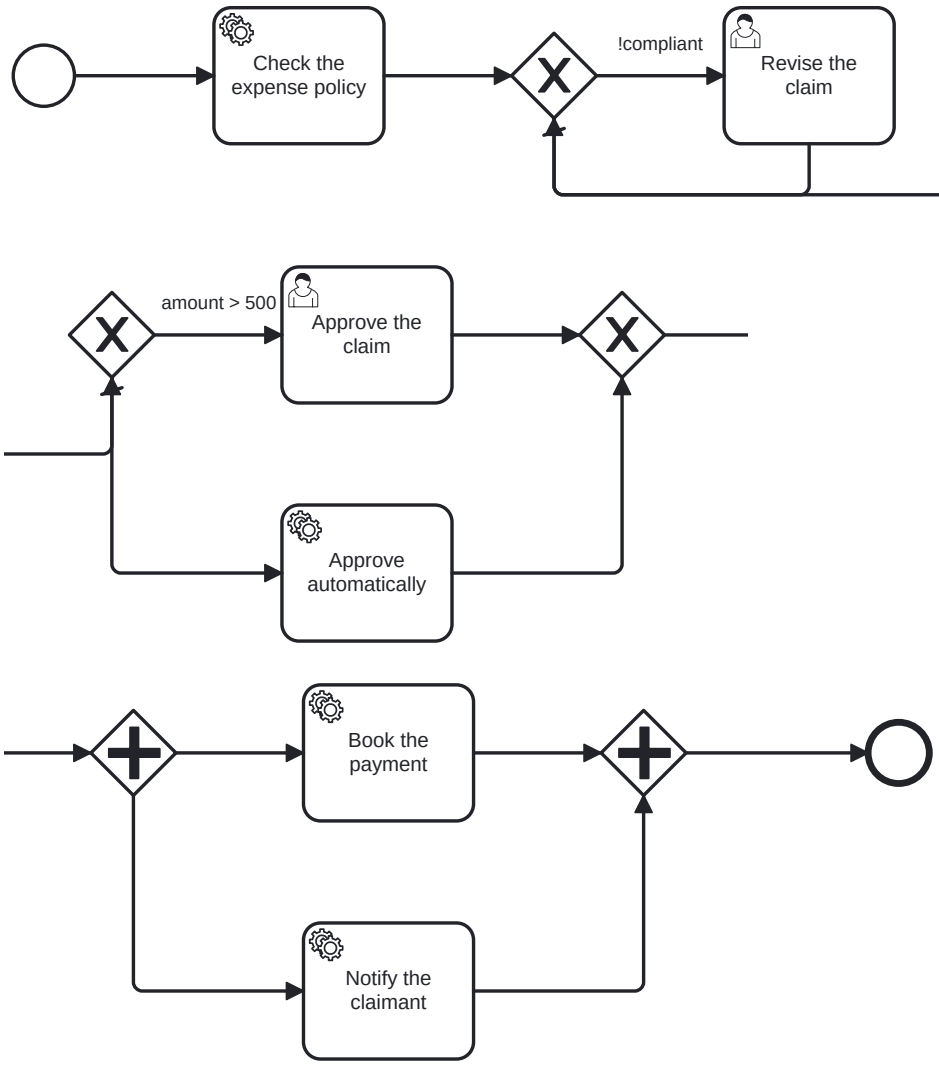


Figure 1.2: The running example of Listing 1.1 after compilation, drawn from the diagram data that `bpmn-auto-layout` 1.3.0 [bpmn.io26] generates, so that it shows the actual output of the compiler. The diagram is split into three rows, and each row ends with a sequence flow that continues at the start of the next row. The line below "Revise the claim" carries two flows: the loop back to the first gateway and the flow to the second gateway once the claim complies. In the second row, the flow from the first row enters the second gateway at the corner from which the flow to "Approve automatically" leaves, and at the first and the second gateway an arrowhead covers the slash of the default flow. The layout library draws these flows on shared lines, which affects only the drawing (Section 5.4.4).

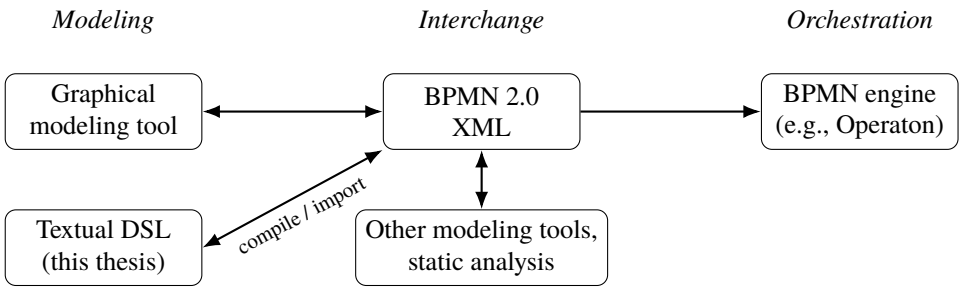


Figure 1.3: The position of the textual DSL: a second notation for the modeling role, connected to interchange and orchestration only through the standardized XML.

file, which means that its parser reads the file and the engine registers the executable processes in it, from which instances can then be started [Ope26a]. What the engine accepts and executes sets the scope of the DSL (Section 4.2). Cross-engine portability is left to future work (Section 7.4.2).

1.3 Research Questions

The overall question of this thesis is how developers can author executable BPMN 2.0 processes as text and keep them interchangeable through the XML-based interchange format. It is broken down into four research questions: RQ1 and RQ2 investigate the problem, RQ3 designs the DSL, and RQ4 validates it.

RQ1: Which textual workflow languages and notations exist, and how does each relate its textual form to a graphical notation and to a workflow engine?

RQ2: Which BPMN 2.0 constructs does Operaton support for execution, and which of them does a developer-oriented textual DSL cover?

RQ3: How can the constructs identified by RQ2 be expressed with structured control flow in a concise textual DSL?

RQ4: Does a round trip from BPMN 2.0 XML through the DSL back to XML preserve the execution-relevant content of executable models?

RQ1 surveys existing notations against R1 to R3, RQ2 sets the scope for R2, RQ3 addresses R1, and RQ4 checks R3 together with R2. In RQ3, a DSL is concise if a process takes fewer non-whitespace characters in it than in each related textual notation in which the same process is written. As a secondary measure, the evaluation compares the size of each example process with that of the BPMN XML it compiles to, without DI. The expense claim of Listing 1.1, for example, takes 745 non-whitespace characters as source and 6,933 as XML. In RQ4, content is execution-relevant if it determines how the engine runs a process, such as the sequence flows, their conditions, and the settings that bind a task to code, but not the diagram layout, which execution does not require [OMG13, Sect. 2.3.1]. The equivalence relation of Section 4.5.4 decides for each

model whether its round trip preserves this content. Section 6.1 describes how RQ4 is answered with this relation.

1.4 Research Method

We follow design science, which iterates over two activities: designing an artifact intended to improve its problem context, and investigating that artifact in context [Wie14, pp. 3–5]. Wieringa [Wie14, pp. 4–5] distinguishes design problems, which ask for an artifact that helps stakeholders, from knowledge questions, which ask for answers about an artifact in context. RQ3 is a design problem. RQ1 and RQ4 are knowledge questions. RQ2 combines a knowledge question about Operaton with the scoping decision based on its answer. The work follows the design cycle, an iteration over problem investigation, treatment design, and treatment validation [Wie14, p. 27], where a treatment is the artifact in interaction with its problem context [Wie14, p. 28].

Problem investigation comprises the review of related work for RQ1 (Chapter 3) and the engine study for RQ2. For RQ2, the element scope is set construct by construct against what Operaton 2.1.0 accepts and executes, read from the source of its parser (Section 4.2). Usage studies of BPMN models serve only as a secondary check of this scope (Section 4.2.3).

We collect the literature, including the related work for RQ1, with keyword searches and snowballing. This is not a systematic literature review, which starts from a protocol that defines the search strategy and inclusion criteria and aims at finding the relevant literature as completely as possible [KC07, Sect. 2.4]. The search starts from the literature of the thesis proposal, which was compiled with the supervisor over several review rounds. Besides these suggestions, the proposal literature comes from keyword searches in Google Scholar, in a general web search, and with the artificial intelligence (AI) research assistant ScienceOS, on topics such as textual notations for BPMN and Business Process Execution Language (BPEL), workflows defined as code, and bidirectional transformations of process models. The web search also finds the documentation and project pages of tools that are not described in any paper. Such sources are grey literature, defined after the Cochrane handbook as “literature that is not formally published in sources such as books or journal articles” [GFM19, Sect. 2.1]. We cite them because several of the textual front ends for BPMN in Chapter 3 exist only in this form. Snowballing starts from such a start set of works [Woh14, Sect. 3.1]: backward snowballing follows the references of each relevant work, and forward snowballing the works that cite it [Woh14, Sect. 3]. Before major design decisions, targeted searches follow: specific keywords, and backward and forward snowballing from the passages of works already found that concern the decision, such as how to translate a graph with almost arbitrary control flow into blocks (Section 2.3).

A work counts as related work for RQ1 if it defines a textual notation for a process or workflow language, or relates such a notation to a graphical process model or to an engine. In addition to reading the abstract, a large language model (LLM) supports the triage of the works by ranking each against the research questions and design problems, from essential to relevant for a single aspect (Appendix I), and a work is read if its abstract or its ranking shows that it is relevant.

Long works that matter for a single aspect of the thesis are read in part: the abstract, the relevant section, and the sections around it. An LLM helps by checking our reading against the full text for misinterpretations, missing context, and further relevant passages. The search is not exhaustive.

Treatment design produces the syntax, the intermediate representation, and the transformations (Chapters 4 and 5). Six candidate syntaxes are drafted as complete renderings of one process and compared analytically before one is chosen (Section 4.3.2 and Appendix D). The design decisions are recorded with their considered options in architectural decision records (ADRs) [KAZ18] (Chapter 4).

Treatment validation checks round trips on generated programs, on publicly available process models, on the official models of Operaton and Camunda, and on reference models written with other modeling tools, and deploys the rebuilt models of the last three on Operaton (Chapter 6). We froze the equivalence relation by which a round trip is judged in release 0.1.1 of BPMNscript [Kra26b] before running the evaluation. For the conciseness part of RQ3, treatment validation counts non-whitespace characters. It compares the DSL with related notations on processes written in both, and each example process with the XML it compiles to (Section 6.5).

Generative AI tools are used for the literature search, the implementation (Section 5.7), and the writing (Appendix I).

1.5 Contributions

The contributions of this thesis are:

- We design a concise, developer-oriented textual syntax that represents the BPMN constructs and control flow needed for executable orchestration processes (RQ2, RQ3; Section 4.3).
- We implement the language with a compiler to BPMN 2.0 XML that generates the diagram layout, an importer that translates BPMN 2.0 XML back into the DSL, and a language server with completion, navigation, rename, hover, and inline diagnostics (Chapter 5). This editor support is a core part of the contribution, since it gives BPMN authoring the benefits of working in code.
- We define the bidirectional transformation, including how import recovers structured statements from a gateway graph, and the equivalence relation by which RQ4 judges a round trip (Section 4.5).
- We evaluate the transformations and the notation with the methods of Section 1.4 and answer RQ4 and the conciseness part of RQ3 (Chapter 6).

1.6 Summary of Results

Results. None of the textual notations found in the related work meets all three requirements of Section 1.1 (Section 3.12). Operaton executes 19 element types and the 54 event combinations that the specification permits, and the DSL has a construct for 17 of these element types and for 52 of these combinations (Section 4.2.4). Three processes have listings in related notations: the invoice approval in three related languages for executable BPMN and in kUML (Sections 3.7 and 3.11), the loan approval in BPELscript (Section 3.1), and the example workflow of the Workflow Execution Engine (WEE) DSL (Section 3.3). On all three, the DSL takes the fewest non-whitespace characters (Section 6.5). Over the 32 example processes of release 0.1.1 (Section 6.5), the XML without DI takes a median of 6.8 times as many non-whitespace characters as the DSL source.

In its default configuration, Operaton refuses a model without a history time to live, the period after which the engine may delete the records of a finished instance. The compiler of BPMNscript therefore adds the default history time to live to every model without one (Section 4.3.5). With the check for this value switched off, Operaton deploys 6,060 executable models mined from GitHub. The importer accepts 4,946 of these models and refuses the other 1,114 with an error that identifies the construct. Of the 4,946 accepted models, 3,858 (78.0 %) round-trip to an equivalent model, and Operaton deploys the rebuild of each, the document compiled from its imported source. Measured against all 6,060 deployed models, the share is 63.7 %. For 3,803 of the equivalent models, the original lacks a history time to live, and the rebuild gains the default. Of the other accepted models, 921 diverge, mostly in their routing, and for the remaining 167 the source that import writes fails the checks that BPMNscript runs before compilation (Section 6.3.4). The share of equivalent models falls with size, from 77.3 % of the deployed models with up to 10 flow nodes to 5.9 % of those with more than 50. Most imported source is structured: the source that import writes for 80.7 % of the accepted models contains no `goto`, and the share with a `goto` grows with size, to 75.5 % above 50 flow nodes (Table 6.11). Of the 2,151 official models of Operaton and Camunda that the importer accepts, 2,015 (93.7 %) are equivalent. All 1,996 valid generated programs round-trip to an equivalent model, and at most four diverge when the elements of the imported model are reversed or shuffled (Section 6.2). Of the 22 MIWG reference models, 2 conform, and the importer refuses the other 20, none of which Operaton deploys (Section 6.4).

Interpretation. For RQ1, the combination of the three requirements is new among the notations we found. For RQ2, the DSL covers all but two of the element types that Operaton executes. Import rewrites the manual task into a plain task, written `step` in the DSL, since Operaton runs it like one. BPMN offers two forms for a compensation handler [OMG13, Sect. 10.7.1], and only one of them fits a block-structured language. Import therefore refuses the other, an association that attaches the handler outside the flow (Section 4.2.4). For RQ3, structured statements express the constructs of RQ2, and `goto` covers the graphs without a block form. On all three processes for which listings in related notations exist, the DSL is the most concise notation. Independently of related notations, every one of the 32 example processes is shorter in the DSL than in the XML it compiles to. For RQ4, the round trip preserves the execution-relevant content of most small

models and of few large ones. The history time to live that most equivalent models gain leaves the run of an instance unchanged (Section 6.6). Models written in the style of the engine, such as the official models, successfully round-trip more often than mined ones. For the overall question of Section 1.3, developers can write executable processes as text that Operaton deploys, and they can import most small models from other tools. The import of large models remains open, and the generated layout replaces the drawing of an imported model (Section 7.4).

1.7 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces BPMN 2.0 in its three roles, experiments on reading text and diagrams, block-structured and graph-based process languages, the gap between the standard and what engines execute, the Operaton engine, language workbenches, and the workflow-as-code paradigm. Textual process and workflow languages are surveyed in Chapter 3, which positions this thesis against them (RQ1). In Chapter 4, we state the design objectives, derive the supported BPMN subset from Operaton (RQ2), and present the syntax (RQ3), the intermediate representation, and the bidirectional transformation with its equivalence relation. The implementation follows in Chapter 5: the architecture, the language front end built with Langium, the editor integration, the compiler, the importer, the command-line tool, the tests, and the use of generative AI. Chapter 6 reports on round-trip correctness on generated programs, mined models, and the official models of Operaton and Camunda, on MIWG interchange conformance and reference models (RQ4), and on the notation size against related notations and against XML (RQ3), and discusses threats to validity. Finally, Chapter 7 answers the research questions (Section 7.2), discusses the limitations, and outlines future work (Section 7.4).

2 Background

This chapter introduces the concepts the thesis builds on. We first present BPMN 2.0 as notation, interchange format, and execution semantics (Section 2.1) and summarize what experiments report on reading text and diagrams (Section 2.2). We then separate block-structured from graph-based process languages (Section 2.3), a difference that import has to bridge (Section 4.5.2). The two sections on engines (Sections 2.4 and 2.5) ground the element scope of Section 4.2. The section on language workbenches (Section 2.6) prepares the choice of Langium in Section 5.2. The chapter closes by contrasting BPMN with workflows defined as code (Section 2.7).

2.1 BPMN 2.0: Notation, Interchange Format, Execution Semantics

The BPMN specification of the Object Management Group (OMG) defines a graphical notation for business processes, a metamodel with an XML serialization, and execution semantics [OMG13, Sects. 7.2, 15.3]. Its stated purpose is a notation that business users understand and that forms "a standardized bridge for the gap between the business process design and process implementation" [OMG13, Sect. 1.1]. This thesis uses the three roles introduced in Chapter 1 (modeling, interchange, and orchestration) in the sense of the notation, interchange format, and execution semantics that the specification defines [OMG13, Sect. 7.2].

This section introduces only the BPMN elements that the examples of the thesis use and assumes the rest of the notation as known. Which element types Operaton executes, and how the DSL supports each of them, is summarized in Table 4.1 and listed in full in Appendices A and B. Figure 2.1 shows a small process in the graphical notation. A start event, a circle drawn with a thin line, marks where the process begins, and an end event, a circle drawn with a thick line, marks where it ends [OMG13, Sects. 10.5.2, 10.5.3]. A task is a rounded rectangle, and a marker in its upper-left corner gives its type [OMG13, Sect. 10.3.3]. A user task, in which a human performs the work with the help of software, carries a human figure [OMG13, Sect. 10.3.3]. A service task, which calls a service, for example a Web service or another automated application, carries a marker of its own, drawn here as gears [OMG13, Sect. 10.3.3]. Sequence flows, the solid arrows, connect the elements and give the order in which they are performed [OMG13, Sect. 8.4.13]. An exclusive gateway is a diamond, here drawn with an X inside [OMG13, Sect. 10.6.2]. Where it splits the flow, it is called diverging: it evaluates the conditions on its outgoing flows in order and continues on the first flow whose condition holds [OMG13, Sect. 13.4.2]. In Figure 2.1, the label on the upper outgoing flow states its condition. A flow with a slash marker at its start is the default flow, which is taken only if the conditions of all other outgoing flows are false [OMG13,

Sect. 8.4.13]. A converging exclusive gateway merges alternative paths, so that whichever path reaches it continues at once, without waiting for the others [OMG13, Sect. 10.6.2].

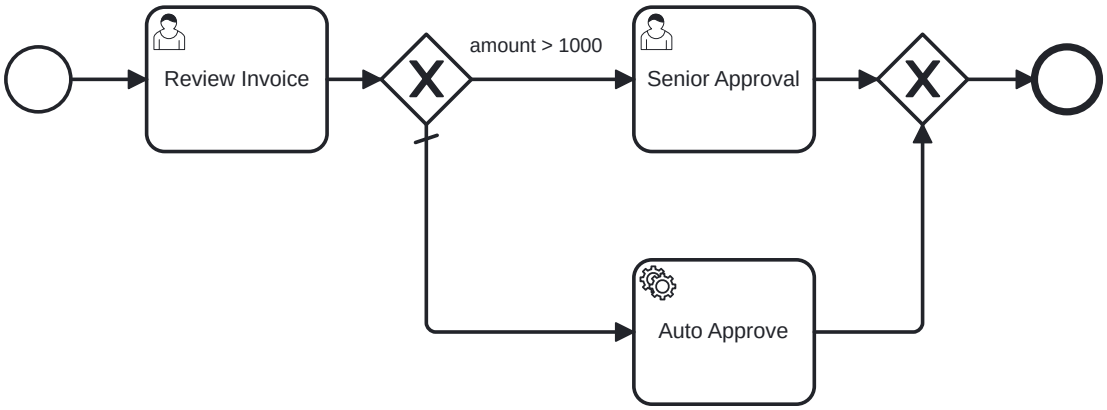


Figure 2.1: A small process in the BPMN notation, the `invoice-approval` example process of BPMN-script [Kra26b]. After the review, an exclusive gateway sends an invoice with an amount above 1000 to a senior approval and every other invoice along the default flow to an automatic approval. A second exclusive gateway merges the two paths.

The interchange role rests on the serialization. The specification states that one of its main goals is an interchange format for both the model and the diagram layout [OMG13, Sect. 2.2.8]. An XML Schema defines the document structure: every file has `definitions` as its root element, and references into other files use qualified names [OMG13, Sect. 15.3]. Layout is kept apart from the model. The Diagram Interchange (DI) is meant to hold only information that is "neither present, nor derivable, from the BPMN model whenever possible" [OMG13, Sect. 12.1], and every diagram plane, shape, or edge references exactly one model element [OMG13, Sect. 12.2]. Execution does not require the diagram: a tool that claims Process Execution Conformance need not support the graphical notation of the specification [OMG13, Sect. 2.3.1]. Rendering a diagram, by contrast, requires both the DI and the model it references [OMG13, Sect. 12.1]. The DSL relies on this separation: its source carries no coordinates, and the compiler regenerates the DI from the model (Section 4.5.3).

The orchestration role rests on the execution semantics. The specification divides an end-to-end model into processes (which it calls orchestrations), choreographies, and collaborations [OMG13, Sect. 7.2.1]. A private process, one internal to a single organization, is either executable or non-executable, and only an executable process is meant to be run by an engine [OMG13, Sect. 7.2.1]. The `isExecutable` attribute marks such a process [OMG13, Table 10.1]. Section 13 of the specification defines the execution semantics for orchestrations [OMG13, Sect. 13.1]. It describes them in terms of tokens, a theoretical concept for a marker that traverses the sequence flows of a process [OMG13, Sect. 7.2.1]. Its subsections cover process instantiation, the activity lifecycle, the behavior of each gateway type, and events, including boundary events, event sub-processes, and compensation [OMG13, Sects. 13.2–13.5]. A choreography, in contrast, has no central controller [OMG13, Sect. 7.2.1]. It is not executed by itself: the services of its participants enact it together at runtime, and each of these services is usually built as an orchestration [DKB08,

Sect. 2]. The DSL therefore writes only executable processes, and its compiler marks each of them with `isExecutable` set to `true` (Section 4.1). Section 2.4 discusses how precisely Section 13 of the specification defines these semantics.

The specification groups its requirements into four conformance types: Process Modeling, Process Execution, BPEL Process Execution, and Choreography Modeling Conformance [OMG13, Sect. 2.1]. A tool claims only the types it needs: Process Modeling Conformance does not require Choreography Modeling Conformance, and Process Execution Conformance requires neither of the two [OMG13, Sect. 2.1]. In terms of the three roles, Process Modeling Conformance covers modeling and, with its clause on visual interchange [OMG13, Sect. 2.2.8], interchange, and Process Execution Conformance covers orchestration. The BPEL type is a special case of Process Execution Conformance that adds the mapping to BPEL [OMG13, Sect. 2.4] and is taken up in Section 2.3.

Full Process Modeling Conformance requires the core elements together with process, collaboration, and conversation diagrams, and a tool can claim one of three smaller sub-classes instead [OMG13, Sect. 2.2.1]. The Descriptive sub-class contains the visible elements of high-level models and addresses analysts familiar with flowcharting tools, and the Analytic sub-class adds further elements until it covers roughly half of what full Process Modeling Conformance contains [OMG13, Sect. 2.2.1]. The Common Executable sub-class addresses modeling tools whose output an engine can execute, and it prescribes XML Schema for data types, the Web Services Description Language (WSDL) for service interfaces, and the XML Path Language (XPath) for data access within the process [OMG13, Sect. 2.2.2]. Whether these sub-classes can serve as the scope of the DSL is examined in Section 4.2.2.

2.2 Text and Diagrams for Readers

This section summarizes experiments that compare how people read textual and graphical representations, and relates them to the developers that the DSL addresses. Petre [Pet95, p. 35] reports experiments on graphical and textual representations of nested conditional structures. Answering questions took significantly longer with the graphical representations than with the textual ones, in every condition, and the mean time with graphics was higher for every single subject [Pet95, p. 35]. She concludes that reading a graphical notation has to be learned, and that what a reader perceives in a representation depends largely on prior training [Pet95, p. 42].

For process descriptions, Ottensooser et al. [OFR+12, p. 597] compared BPMN diagrams with written use cases, a structured textual notation. Readers with and without training in process modeling both understood a process significantly better after reading the use cases than from their background knowledge alone [OFR+12, pp. 602–603]. After reading the diagram, this gain was significant only for the trained readers [OFR+12, pp. 602–603]. For the untrained readers, however, the difference between the use cases and the diagram was not significant [OFR+12, pp. 599, 602]. Ivanchikj et al. [ISP22, pp. 1895, 1897] report other experiments that favor the

graphical notation for some tasks, such as recalling the content of a class diagram or making sense of a declarative process model.

Genon et al. [GHA11, pp. 377, 379–380] analyze the visual notation of BPMN 2.0 itself, in a draft of the specification (version 0.9.15), against the nine principles of the Physics of Notations, a theory of notation design built from empirical evidence. They count 171 symbols in process diagrams, about 28 times the roughly six visual alternatives that people can tell apart by absolute judgement [GHA11, pp. 383, 392]. Since novices find it harder than experts to tell symbols apart and to remember what they mean, the authors consider the notation more challenging for novices [GHA11, p. 382]. The developers targeted in Chapter 1 are such novices, since they have not learned the BPMN notation.

From the developer's viewpoint, Grönniger et al. [GKR+07, Sect. 2] argue that textual models show more detail per screen, need no switching between mouse and keyboard, and merge line by line in version control. They concede that a graphical model gives the quicker overview [GKR+07, Sect. 2]. As with diagrams, how well a developer reads code depends on training [Pet95, p. 42], and the developers targeted in Chapter 1 are trained in reading code.

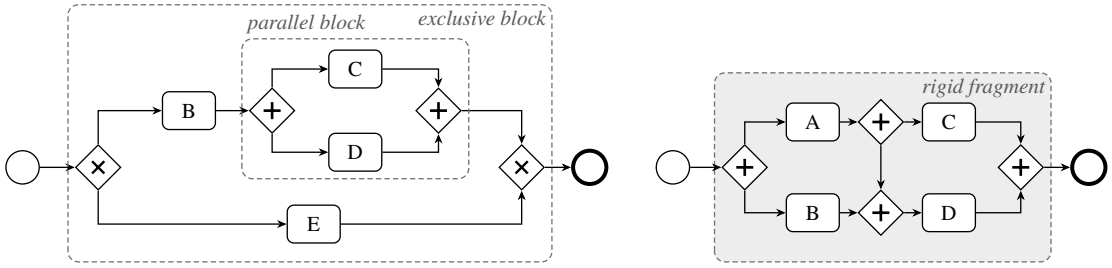
2.3 Block-Structured and Graph-Based Process Languages

This section separates block-structured from graph-based process languages, states when a graph has a block-structured equivalent, and describes how the blocks of a graph are found, with the translation from BPMN to BPEL as an earlier instance of the problem. Kopp et al. [KMWL09, p. 3] distinguish block-structured languages, which define control flow with nested constructs such as `if` and `while` as programming languages do, from graph-based languages, which connect activities by explicit control links. Languages that combine both approaches form a third, hybrid class [KMWL09, pp. 3, 7]. In BPMN 1.2, the version Kopp et al. [KMWL09, pp. 3, 9–10] analyze, models are graph-based and allow arbitrary cycles, and the only block constructs are a `while` loop and the sub-process. BPMN 2.0 keeps this character: according to its specification, a modeler can draw nearly any graph as control flow [OMG13, Sect. 14.1]. A mapping between the two classes exists in both directions, provided the block-structured side emulates arbitrary cycles with its own constructs [KMWL09, p. 11]. Because a model can be mapped in more than one way, and because of this emulation, a model translated to the other class and back usually differs in structure from the original [KMWL09, p. 11]. Import and compilation of the DSL form such a round trip, so an equivalence relation decides whether the returned model matches the original (Section 4.5.4).

Without such emulation, a graph-based model can be translated into blocks only if it has a block-structured equivalent. Kiepuszewski et al. [KHB00, Def. 1] define structured workflow models inductively: a single activity, the sequence of two structured models, structured branches between a split and a join of the same kind, and a loop between a join and a split. Every unstructured workflow without parallelism has an equivalent structured form [KHB00, Theorem 1], although some forms of unstructuredness can be removed only with auxiliary variables [KHB00, Theorem 3]. With

parallelism this no longer holds: some models are free of deadlocks and never run an activity in several instances at once, yet have no structured form [KHB00, Def. 2, Theorem 2]. Where a graph has no structured form, or has one only with auxiliary variables, the printer of the DSL, the part of import that writes source, falls back to `goto` (Section 4.5.2).

To print an imported graph as blocks, a structuring method first has to find the parts of the graph that can become blocks, the fragments with a single entry and a single exit. The Refined Process Structure Tree (RPST) of Vanhatalo et al. [VVK09, p. 793] arranges these fragments of a workflow graph in a tree that is unique, modular, and computable in linear time. Polyvyanyy et al. [PVV11, p. 25] simplify its computation and extend it to graphs with several sources and sinks. The tree classifies each fragment as a single flow, a sequence of fragments, a set of fragments between the same entry and exit, or, if it is none of these, a rigid fragment [PGD12, pp. 520–521, Def. 2.3]. A model is structured exactly when its RPST contains no rigid fragment [PGD12, p. 521, Def. 2.4]. Every rigid fragment contains a split or join without a matching partner [PGD12, p. 521]. Figure 2.2 contrasts a structured model, whose fragments nest as blocks, with the model of Kiepuszewski et al. [KHB00, Theorem 2], whose fragment between the first split and the last join is rigid. Such fragments can also be found through dominance: a node d dominates a node n if every path from the start to n passes through d, and a node p post-dominates n if every path from n to the end passes through p [CHKH20, p. 616]. Choi et al. [CHKH20, Def. 2] define a single-entry single-exit region by interior nodes that its entry dominates, its exit post-dominates, and that connect only to nodes of the region, and they give a region of this kind that the existing RPST methods miss [CHKH20, pp. 614, 616]. The printer of the DSL finds the blocks of an imported graph through dominators and post-dominators (Section 4.5.2). A printer built on the RPST is future work (Section 7.4.6).



(a) Structured (b) Unstructured

Figure 2.2: Two BPMN process models, with single-entry single-exit fragments marked as dashed boxes. (a) A structured model in the sense of Kiepuszewski et al. [KHB00, Def. 1]: a parallel block nests inside one branch of an exclusive block, and each block starts with a split and ends with a join of the same kind. (b) An unstructured model after Polyvyanyy et al. [PGD12, p. 521, Fig. 5], which draws the counter-example of Kiepuszewski et al. [KHB00, Fig. 6, Theorem 2] in BPMN: the split after A also leads into the join before D, so the fragment from the first split to the last join is rigid, and the model has no structured equivalent.

The translation from BPMN to BPEL is an earlier instance of this structuring problem. When Ouyang et al. [ODHA06, p. 285] wrote, no engine executed BPMN models directly, whereas several platforms ran BPEL, so the authors saw a mapping from BPMN to BPEL as a necessary

step toward standards-based process development. BPEL 2.0, a standard of the Organization for the Advancement of Structured Information Standards (OASIS), specifies how business processes built only from Web services and XML data are executed, in a form that is portable between platforms [AAA+07, Sect. 1]. Its structured activities, such as `sequence`, `if`, and `while`, nest as blocks, and links inside a `flow` add explicit control dependencies between concurrent activities [AAA+07, Sects. 5.2, 11, 11.6]. The specification calls this "a blending of graph-structured and block-structured modeling styles" [AAA+07, Sect. 11], and Kopp et al. [KMWL09, p. 7] classify BPEL as a hybrid language. Not every BPMN model maps directly onto these constructs, because BPEL allows control flow only in blocks or in graphs without cycles [OMG13, Sect. 14.1; ODHA06, p. 285]. An unstructured loop, for example, needs an indirect encoding in BPEL [OMG13, Sect. 14.1]. Ouyang et al. [ODHA06, pp. 287–288] decompose a model into fragments with a single entry and a single exit and map the well-structured ones onto structured activities. The remaining fragments, such as unstructured loops, become event-action rules, which yields less readable BPEL [ODHA06, p. 289]. The printer of the DSL faces the same task with the statements of the DSL as its blocks, and where Ouyang et al. [ODHA06] fall back to event-action rules, it falls back to `goto` (Section 4.5.2). BPMN 2.0 added execution semantics and a standardized serialization, so a model can run without a prior translation step [GHLW18, pp. 250–251]. Its mapping to BPEL remains as Section 14, required only for BPEL Process Execution Conformance [OMG13, Sects. 2.4, 14.1]. BPELscript, a textual syntax for BPEL [BKLL09], belongs to this BPEL layer and is discussed in Section 3.1.

2.4 Engine Conformance Gap

This section describes what the specification leaves to the engines and how far open-source engines implement it. BPMN 2.0 defines its execution semantics in prose: the specification describes them "informally (textually)," based on earlier research that formalized execution semantics in mathematical terms [OMG13, Sect. 13.1]. Geiger et al. [GHLW18, p. 251] report earlier criticism that this description is imprecise and ambiguous in places. Several further aspects are left to the individual engine. The specification uses XML Schema for data types, XPath for expressions, and WSDL for service operations [OMG13, Sect. 7.2]. A model may replace each of them: the type and expression languages are defaults that can be overridden [OMG13, Table 8.1], and a service task may specify an implementation technology other than Web services [OMG13, Table 10.8]. For script tasks, the specification prescribes no language and only requires that the script format be stated [OMG13, Table 10.12], and an implementation may ignore any extension that is not declared mandatory [OMG13, Sect. 7.7].

Engines fill this open space with their own choices. Geiger et al. [GHL+15, pp. 24–25] tested three open-source engines, jBPM 6.0, Activiti 5.16, and Camunda BPM 7.1, with 70 feature tests for 27 language constructs. Activiti and Camunda BPM execute scripts and evaluate conditions in Groovy, jBPM in Java [GHL+15, p. 26]. None of the three engines calls or provides Web services through the WSDL-based interfaces that the standard requires: each exchanges messages through its own mechanism, so message events and service, send, and receive tasks could not

be tested [GHL+15, p. 29]. All three engines support 30 of the 70 tested features (43 %), and the authors regard only this shared set as fully portable among them [GHL+15, p. 28; Len16, p. 92]. Even exclusive gateways are not portable without restriction, since jBPM does not support a gateway that both merges and splits the flow [GHL+15, p. 28]. For this reason, the DSL targets a single engine (Section 1.2).

Geiger et al. [GHLW18, pp. 250, 252–253] screened 47 products that claim to conform to BPMN 2.0 and found only three engines, Activiti, Camunda BPM, and jBPM, that execute the standard serialization and can be deployed and run without manual steps, as their automated benchmark requires. Over more than three years of releases up to June 2016, feature support across the three engines grew by only 19 features [GHLW18, pp. 250, 253, 258]. The engine with the highest support, Camunda BPM 7.5, supported 57 % of the tested language features [GHLW18, pp. 256, 260]. The authors read this stagnation as a sign that, from the implementers' perspective, the implementation of the standard "has largely been concluded at the current level" [GHLW18, p. 260]. These results concern Camunda BPM 7.0 to 7.5 [GHL+15, p. 25; GHLW18, p. 259], the predecessors of the engine this thesis targets (Section 2.5). The support in Operaton itself is read from the source of its parser (Section 4.2.1).

2.5 The Operaton BPMN Engine

Operaton [Ope25] is the BPMN engine targeted by this thesis, a target set as part of the thesis goal. It is an open-source BPMN 2.0 engine under the Apache License 2.0 [Ope26a].

Operaton continues a code base with a long history, spanning Activiti 5, Camunda BPM 7, and Operaton itself. Camunda BPM 7 started in 2013 as a fork of Activiti 5 [Har17, pp. 64–65], so the Camunda BPM releases benchmarked in Section 2.4 are earlier releases of the code line that Operaton continues. Operaton was forked in 2024 from this code base, by then released as Camunda 7, with a first beta release in October 2024 and a first stable release, 1.0.0, in November 2025 [Ope26b]. Camunda announced that the Community Edition of Camunda 7 ends with release 7.24, scheduled for October 2025, and that no further releases, including security patches, follow [Cam25]. Operaton reached compatibility with Camunda 7.24 by backporting the upstream releases 7.23 and 7.24 [Ope26b]. Three properties of Operaton support this target. It continues as open source the code line of one of the three engines that Geiger et al. [GHLW18, p. 253] could benchmark (Section 2.4). Camunda is among the modeling tools that Türker et al. [TVH22b, p. 70] list as popular for designing BPMN models. Because Operaton is open source, the element scope of the DSL can be read from the source of its deployment parser (Section 4.2). All deployments in this thesis, in the tests and in the evaluation, run on Operaton 2.1.0 [Ope26a].

Operaton deploys and executes BPMN 2.0 XML directly [Ope26a]. On a new deployment, the engine registers only the processes whose `isExecutable` attribute is `true` and skips every other process of the file [Ope26a]. What the standard leaves to the engine (Section 2.4) is configured in the extension namespace `operaton:` (`http://operaton.org/schema/1.0/bpmn`): the binding of a service task to code, forms, and further engine settings that Section 4.3.5 introduces [Ope26a].

For compatibility with existing models, the parser falls back to the Camunda extension namespace `camunda:` for every extension attribute or element missing from `operaton:` [Ope26a].

Deployment alone does not guarantee correct execution. Earlier releases of the same code line accepted unsupported constructs: of the 45 tested features that Camunda BPM 7.5 did not support, it rejected only 18 at deployment and executed the other 27 incorrectly at runtime [GHLW18, p. 256]. For this reason, the scope of the DSL counts a construct as supported only if the engine also executes it (Section 4.2).

2.6 Language Workbenches

This section introduces domain-specific languages, the ways to implement them, and the workbenches and the protocol that give them editor support. Mernik et al. [MHS05, pp. 316–317] characterize a DSL as a language built for one application domain, which gives up generality to become more expressive within that domain. Karsai et al. [KKP+09, Sect. 2] collect 26 guidelines for designing a DSL, grouped into its purpose, realization, content, concrete syntax, and abstract syntax, and Section 4.3.2 applies several of them to the syntax of the DSL. One way to implement a DSL is embedding: its constructs become new abstract data types and operators of an existing general-purpose language, the host language [MHS05, p. 329]. An embedded DSL reuses the editors and debuggers of its host, but its syntax is limited to what the host allows, and its error messages are phrased in terms of the host language [MHS05, p. 331]. Voelter [Voe13, pp. 42, 50] calls such languages internal DSLs, in contrast to external DSLs, for which an integrated development environment (IDE) with language-specific code completion and static analysis can be built. DSLs often lack the editor support that IDEs offer for general-purpose languages, such as syntax highlighting, navigation to definitions, and error highlighting [CHR26, p. 123]. Implementing a DSL with a compiler or interpreter of its own is costly, since a complete language processor has to be built [MHS05, p. 331]. Mernik et al. [MHS05, p. 331] note that this and the other disadvantages of the approach "can be minimized or eliminated altogether when a language development system or toolkit is used." Language workbenches are such toolkits: "tools that provide high-level mechanisms for the implementation of (domain-specific) languages" [ESV+13, p. 197]. Parser-based workbenches derive the parser and the construction of the abstract syntax tree (AST) from a grammar [Voe13, p. 68; SS25], so that the parser need not be written and kept in sync with the grammar by hand. Most workbenches also provide editor services such as syntax coloring, outlines, and code completion, and some validate models [ESV+13, Sect. 3].

Integrating every language into every IDE separately requires one integration per pair of language and IDE [Bün19, p. 131]. The Language Server Protocol (LSP), introduced in 2016, decouples editor features such as code completion and validation from the editor that displays them [Bün19, p. 130]. A language server computes these features in its own process and answers a client in the editor over a protocol based on JSON-RPC [Bün19, p. 131]. The language-specific part is thus implemented once and serves every editor that has an LSP client [Bün19, p. 137; CHR26, p. 125]. BPMNscript provides such a language server (Section 5.3).

Eclipse Xtext [Ecl26] generates this infrastructure for the Java Virtual Machine (JVM), with editors for Eclipse and for any client of the LSP [Ecl26; CHR26, p. 126]. Zhang and Strüber [ZS24, p. 173] consider it the dominant technology for textual DSL development in model-driven engineering. Langium [Langium26a] follows the same grammar-first approach: Zhang and Strüber [ZS24, p. 172] call Xtext a blueprint for it, and its developers place both in the same tradition of language engineering [SS25]. Langium is written in TypeScript and built around the LSP [Langium26a; CHR26, p. 127]. Its command-line generator derives the AST types, a serialized grammar, and, if configured, a TextMate grammar for syntax highlighting from the grammar file [Langium26e]. Its runtime turns them into a language server that runs in Node.js or in a browser [Langium26a; CHR26, p. 127].

MontiCore [RH17] relies on Java [CHR26, p. 127] and composes languages through inheritance, embedding, and aggregation [RH17, pp. 81–82]. It can also generate a language server, together with plugins for IntelliJ and Visual Studio Code, from a grammar [CHR26, p. 123]. The textual BPMN notation of Drave et al. [DMM+22], built with MontiCore, is discussed in Section 3.4. JetBrains MPS takes a different route: as a projectional workbench, its editor changes the AST itself, so a language needs neither grammar nor parser [Voe13, p. 68]. Which workbench this thesis uses, and why, is argued in Section 5.2.

2.7 The Workflow-as-Code Paradigm

Several workflow platforms define processes directly in a general-purpose programming language. Apache Airflow calls this the “workflows as code” approach, credits it with dynamic pipeline generation and extensibility, and lists version control and testing among the benefits of defining workflows as Python code [Airflow24]. In Temporal, a workflow is defined in code written with one of its language software development kits, typically as a function or an object method [Temporal24]. These platforms gain the tooling of the host language, but their documentation defines neither a standardized visual notation for workflows nor an interchange format between modeling tools and engines [Airflow24; Temporal24]. BPMN has both, and the BPMN MIWG defines test cases for the interchange of models between tools [OMG26]. The DSL of this thesis sits between these platforms and BPMN: its source is text, but its compiler writes BPMN 2.0 XML, so a process written in it keeps the notation and the interchange format that the workflow-as-code platforms lack (Section 3.12). Other textual front ends for BPMN and for neighboring process languages are reviewed in Chapter 3.

3 Related Work

This chapter answers RQ1 (Section 1.3) by reviewing textual process and workflow languages and how each relates its text to graphical process models and to workflow engines. Sections 3.1 to 3.3 cover languages whose primary target is not BPMN. Textual notations and code front ends for BPMN and related notations follow in Sections 3.4 to 3.7. We then turn to editors that mix text and diagrams and to tools that generate BPMN from natural language (Sections 3.8 and 3.9). Section 3.10 covers transformations between block-structured and graph-based process languages and the theory of bidirectional transformations. We write the loan approval of BPELscript and the example workflow of the WEE DSL in the DSL (Sections 3.1 and 3.3). Each translated listing compiles, and import prints the same statements back. We also write one process in the DSL and in three related languages for executable BPMN (Section 3.11) and compare the textual languages (Table 3.2). From this comparison, Section 3.12 derives the gap stated in Section 1.1. Chapter 4 designs a language that closes this gap.

3.1 BPELscript and the BPEL Precedent

BPELscript [BKLL09] is a direct precedent for this thesis. It gives BPEL 2.0 a syntax inspired by JavaScript, to lower the barrier to BPEL for IT experts, who expect a Java-like syntax [BKLL09, p. 39]. In the lifecycle it targets, a BPMN model is exported to BPEL and converted to BPELscript [BKLL09, p. 39]. An IT expert completes the technical details in an IDE, and the result is converted back to BPEL for deployment [BKLL09, p. 39]. Since BPELscript covers every BPEL construct, the mapping works in both directions [BKLL09, p. 39]. In the authors' view, the only other language close to BPEL in compatibility is SimPEL [BKLL09, pp. 40–41]. Its developers, the Apache ODE team, "preferred language consistency over BPEL compatibility," so SimPEL runs only on their engine, Apache ODE [BKLL09, pp. 40–41]. BPELscript translates fully to BPEL and deploys to any BPEL engine [BKLL09, p. 41].

Structured activities become braced blocks, and graph links are written as `signal` and `join` statements [BKLL09, p. 42]. The authors note that a graph in textual syntax amounts to programming with `goto` [BKLL09, p. 42]. Because `signal` and `join` allow parallel branches, such a graph cannot always be rewritten as nested blocks [BKLL09, p. 42; KHB00, Theorem 2]. In their example, the loan approval process of the BPEL specification, the XML has 90 lines and the BPELscript listing of the full process has 48 [BKLL09, pp. 40–41, Fig. 4]. The translator is built with ANother Tool for Language Recognition (ANTLR) and translates in both directions through abstract syntax trees [BKLL09, pp. 44–45]. Since a process is remodeled in every lifecycle round, the authors

propose to compute the difference between two generated versions as an edit script [BKLL09, p. 45]. Applying this edit script to the next generated version gives a starting point that keeps the details the developer added during completion and also the developer's formatting [BKLL09, p. 45]. This works as long as the operations of the script still apply to the changed model [BKLL09, p. 45]. An IDE with syntax highlighting and syntax checking was ongoing work, and an evaluation of the syntax against editing XML directly was still to come [BKLL09, p. 45].

The DSL also writes control flow as braced blocks, but it targets BPMN 2.0: its blocks compile into the gateways and sequence flows of a graph-based language, whereas those of BPELscript map onto the structured activities of BPEL. The DSL covers most of the subset of the standard that its engine executes (Section 4.2.4). Like SimPEL, it is bound to one engine, but its output stays standard BPMN 2.0 XML with engine extensions. Its import regenerates the source from the XML and does not patch an earlier version, so the formatting of a previous source is lost (Section 3.10). It provides the editor support that BPELscript reported as ongoing work.

Listing 3.1 reproduces the loan approval of Bischof et al. [BKLL09, Fig. 4] without its declarations and its fault handler. Listing 3.2 writes the same process in the DSL. In BPELscript, the process is a `parallel` block of five branches, and `signal` and `join` statements on links put them in order [BKLL09, pp. 41–42]. Two activities wait for two links each, in lines 33 and 38. The approval follows the receive for an amount of 10,000 or more and the risk assessment for a risk other than low. The reply follows the approval and the assignment that accepts the loan. The DSL writes this graph with two nested `if` statements and one `goto`. The jump takes the accepted loan past the approval to the reply. Without it, a second `if` has to test the amount again. Each activity of the BPELscript listing becomes one statement: the receive a message start event, the assignment a script task, and the reply a send task.

3.2 StalkCD: CI/CD Pipelines Between Jenkinsfiles and BPMN

StalkCD [DKH21] is a model-driven framework for continuous integration and continuous delivery (CI/CD) pipelines. Its DSL, a YAML Ain't Markup Language (YAML) serialization of a pipeline metamodel, is the intermediate representation between Jenkinsfiles and BPMN, and its transformations run in both directions [DKH21, p. 217]. A pipeline is a sequence of stages, and a stage may run its nested stages in parallel or run only under a condition [DKH21, p. 217]. The transformation to BPMN derives the gateways from these stages, for example an exclusive gateway pair for a conditional stage [DKH21, p. 219]. Jenkinsfiles are parsed with an ANTLR grammar [DKH21, p. 218], and the BPMN output serves visualization and analysis and is "not required to be executable by a general-purpose BPMN engine" [DKH21, pp. 214, 218]. Its evaluation is a precedent for Section 6.3. The authors translated 1,027 Jenkinsfiles mined from GitHub to their DSL and back and compared them as text after nine normalization steps, which remove formatting such as comments, whitespace, and the quotation marks of stage names [DKH21, pp. 220–221]. Of these, 717 (70 %) came back without information loss [DKH21, pp. 220–221]. The DSL of this thesis produces BPMN that executes on a concrete engine, and it is a language with its own grammar and editor support.

List. 3.1: The loan approval in BPELscript, an excerpt of Bischof et al. [BKLL09, Fig. 4] with its line numbers. The excerpt leaves out the declarations of namespaces, WSDL import, and partner links and the annotation `@suppressJoinFailure` (lines 1–6 and 8–11), and the `try` block with its fault handler (lines 12 and 42–47).

```
 7 process pns::loanApprovalProcess {

13   parallel {
14     @portType lns::loanServicePT @createInstance
15     request = receive(customer, request);
16     signal(receive-to-assess,
17       [$request.amount < 10000]);
18     signal(receive-to-approval,
19       [$request.amount >= 10000]);
20   } and {
21     join(receive-to-assess);
22     @portType lns::riskAssessmentPT
23     risk = invoke(assessor, check, request);
24     signal(assess-to-setMessage,
25       [$risk.level = 'low']);
26     signal(assess-to-approval,
27       [$risk.level != 'low']);
28   } and {
29     join(assess-to-setMessage);
30     approval.accept = "yes";
31     signal(setMessage-to-reply);
32   } and {
33     join(receive-to-approval, assess-to-approval);
34     @portType lns::loanApprovalPT
35     approval = invoke(approver, approve, request);
36     signal(approval-to-reply);
37   } and {
38     join(approval-to-reply, setMessage-to-reply);
39     @portType lns::loanServicePT
40     reply(customer, request, approval);
41   }

48 }
```

3.3 Process Languages Outside BPMN

Several workflow systems outside BPMN define their processes as text from the start. The Cloud Process Execution Engine (CPEE) [MR14] executes processes on an engine whose core is the WEE DSL, an internal process DSL hosted in Ruby [MSS10, pp. 1–2; VPMR25, p. 191]. The web-based Process Design user interface of CPEE edits these processes graphically, and its Process Repository keeps each saved model under version control in Git [MR22, p. 17]. BPMN 2.0 models are accepted as input and translated into the native language of the engine [MR22, pp. 5, 26–27]. The WEE DSL is block-structured [MSS10, pp. 7, 14]. It has constructs for sequence, a guarded

List. 3.2: The loan approval of Listing 3.1 in the DSL. The script body starts at the first column, since the text between the fences reaches the engine as written.

````
process loan-approval {
  var amount: number
  var riskLevel: string

  start RequestReceived message("LoanRequest")
  if (amount < 10000) {
    service AssessRisk(class: "com.example.loan.AssessRiskDelegate")
    if (riskLevel == "low") {
      script AcceptLoan ```groovy
execution.setVariable("accept", "yes")
```
      goto ReplyToCustomer
    }
  }
  service ApproveLoan(class: "com.example.loan.ApproveDelegate")
  send ReplyToCustomer(class: "com.example.loan.ReplyDelegate")
  end Done
}
````

choice (`choose` with `alternative` branches and an optional `otherwise`), parallel branches, and top-controlled loops (`cycle`) [MSS10, pp. 1–2]. Its authors report what the block structure cannot express [MSS10, p. 7]. The local and the general synchronizing merge are two workflow patterns for unstructured processes: they continue once every branch has either arrived or is known to never arrive at the merge [MSS10, p. 7]. Both "cannot be translated into the WEE-DSL as the process definition is expressed in a block oriented manner" [MSS10, p. 7]. Arbitrary cycles cannot be expressed directly either, and are simulated by the handler of the engine, which jumps back to a position given in the workflow description [MSS10, pp. 15–16]. HiProX [VPMR25] executes CPEE processes on resource-constrained devices: it transpiles a process model into a program in a Rust-hosted DSL based on the Ruby one and compiles it natively [VPMR25, pp. 185, 188, 191].

CPEE is close to this thesis in one respect: both give the author block-structured control flow, with no explicit graph. The two languages share a core of sequence, service calls, scripts, choice, parallel branches, and loops [MSS10, pp. 1–2]. On this core, each statement of one language has a counterpart in the other. A WEE call activity becomes a service task, and a WEE manipulate activity becomes a script task [MSS10, p. 2]. A `parallel` block with its `parallel_branch` blocks becomes a `parallel` block [MSS10, p. 2]. A `cycle` becomes a `while` loop, and a context variable becomes a declared variable [MSS10, p. 2]. A `choose` block whose guards exclude each other becomes an `if` statement. If several guards hold, `choose` runs the chosen alternatives one after another [MSS10, p. 5]. The DSL has no form for this sequential choice, since an inclusive gateway starts its branches concurrently [OMG13, Sect. 13.4.3]. For the concurrent variant, the WEE DSL encloses `choose` in a `parallel` block and the body of each alternative in a `parallel_branch` [MSS10, p. 5]. This variant becomes a `parallel` block with conditioned branches, which compiles to an
````

inclusive gateway pair (Table 4.3). Listing 3.3 reproduces the example workflow of Mangler et al. [MSS10, Listing 1], which books and pays a flight and a hotel for each person. Listing 3.4 writes it in the DSL by this mapping. Its `choose` block in lines 44–48 has a single alternative, so it becomes an `if` statement. The translation leaves out the two critical sections in lines 20 and 30, blocks of parallel branches that may not run at the same time [MSS10, p. 2], since the DSL has no critical section. The endpoints, the booking data, and the code that adds up the prices move into the delegate classes, the Java classes that Operaton calls for a service task (Section 4.3.5).

Beyond the shared core, only the WEE DSL has critical sections and a canceling partial join, which continues after a given number of branches and stops the others [MSS10, pp. 2, 6]. In BPMN, a partial join is a complex gateway [OMG13, Sect. 10.6.5], and Operaton fails the deployment when a sequence flow leads to or from one (Table 4.1). Only the DSL has constructs for events, compensation, and `goto` (Table 4.3). For arbitrary cycles, the WEE DSL relies on the handler described above [MSS10, pp. 15–16], and the DSL has a `goto` statement that its validator checks (Section 5.2.3). From the DSL to CPEE, the compiled XML is a BPMN 2.0 model, a form of input that CPEE accepts [MR22, pp. 5, 26–27]. We have not tested this route. The DSL is an external language with its own grammar, validation, and language server, and it compiles to BPMN 2.0 XML, a standard specified independently of the engine that runs it.

HiProX chooses the host language of its internal DSL by three criteria [VPMR25, p. 190]. One of them asks that their DSL read clearly and that its code take little effort to change, so that the code can mediate between the graphical model and its textual form [VPMR25, p. 190]. The DSL of this thesis has the same aim but is an external language, whose own syntax and language server provide the editor support that a host language brings to an internal DSL (Section 2.6).

Topology and Orchestration Specification for Cloud Applications (TOSCA) describes the components of a service and their relationships as a topology, and its workflows order the operations that create or change such a service [LC25, Sect. 1]. TOSCA specifies imperative workflows as YAML steps with explicit successors through `on_success` and `on_failure`, and it leaves declarative workflows for further study [LC25, Sect. 13.1–13.2]. Instead of listing steps, a TOSCA workflow may point to an artifact written in another workflow language [LC25, Sect. 13.2]. The workflow syntax of GitHub Actions defines a workflow as a YAML file [GH26]. Neither language reads or writes BPMN itself [GH26; LC25]. Each TOSCA step states its successors, so the author writes the workflow graph edge by edge. The DSL derives the edges from nested statements, keeps a standardized visual notation beside the text, and targets an engine that executes BPMN 2.0.

3.4 Textual Notations and Code Front Ends for BPMN and Related Notations

Textual notations have also been proposed for graphical process languages in general. TN4PM covers the most important elements of BPMN 1.1, Unified Modeling Language (UML) activity diagrams, and role activity diagrams, and it also serves to translate a model from one of these notations into another [MU09, pp. 263–264]. TN4PM takes jump labels for activity flows from the

List. 3.3: The example workflow of the WEE DSL, reproduced from Mangler et al. [MSS10, Listing 1].

```
1  class Workflow < Wee
2   handler MyHandler
3
4   endpoint :epAirBook => "uri:air/booking"
5   endpoint :epHotelBook => "uri:hotel/booking"
6   endpoint :epAirPay => "uri:air/payment"
7   endpoint :epHotelPay => "uri:hotel/payment"
8   endpoint :epInform => "uri:company/inform"
9
10  context :persons => 3
11  context :creditcard => "Visa_12345"
12  context :airline => nil, :hotel => nil
13  context :from => "Vienna", :to => "Prag"
14  context :sum => 0
15
16  control flow do
17   cycle(@persons > 0) do
18    parallel(:wait) do
19     parallel_branch do
20      critical(:airbooking) do
21       activity :bookFlight, :call, epAirBook, @from, @to do |id|
22        @airline = id
23       end
24       activity :payFlight, :call, epAirPay, @airline, @creditcard do |amount|
25        @sum += amount
26       end
27      end
28     end
29     parallel_branch do
30      critical(:hotelbooking) do
31       activity :bookHotel, :call, epHotelBook, @to do |id|
32        @hotel = id
33       end
34       activity :payHotel, :call, epHotelPay, @hotel, @creditcard do |amount|
35        @sum += amount
36       end
37      end
38     end
39    end
40    activity :countdown, :manipulate do
41     @persons -= 1
42    end
43   end
44   choose do
45    alternative(@sum > 10000) do
46     activity :inform, :call, epInform, @sum
47    end
48   end
49  end
50 end
```

List. 3.4: The example workflow of Listing 3.3 in the DSL, without its critical sections, endpoints, and booking data.

````
process trip-booking {
  var persons: number
  var sum: number

  start TripRequested
  while (persons > 0) {
    parallel {
      {
        service BookFlight(class: "com.example.trip.BookFlightDelegate")
        service PayFlight(class: "com.example.trip.PayFlightDelegate")
      }
      {
        service BookHotel(class: "com.example.trip.BookHotelDelegate")
        service PayHotel(class: "com.example.trip.PayHotelDelegate")
      }
    }
    script Countdown ```groovy
execution.setVariable("persons", persons - 1)
```
  }
  if (sum > 10000) {
    service Inform(class: "com.example.trip.InformDelegate")
  }
  end TripBooked
}
````

simulation language GPSS/H [MU09, p. 264]. Drave et al. [DMM+22, p. 13] mention if-then-else as a further construct, so TN4PM combines structured statements with jumps. The DSL has both constructs as well, but expresses control flow mainly as structured statements, and its importer prints a `goto` only for an edge without a block form (Section 4.5.2). EPClets is a textual language for event-driven process chains (EPCs) that describes a process through declarative rules, each linking events to an action, and constructs the graph from them automatically [KHCR14, p. 693]. A description contains no graph information: each function has a pre- and a post-condition over events, and the tool inserts connectors, the gateways of EPCs, when it links the graphs of functions that share an event [KHCR14, pp. 695–696]. Its Eclipse editor is built with Xtext and shows the constructed graph beside the text [KHCR14, p. 697]. In a controlled experiment with eight participants, modeling a process with EPClets took on average 331 of the 868 seconds needed with a graphical EPC tool, or 38 % [KHCR14, pp. 697–698, Table I]. Participants repeatedly inspected the generated graph while writing [KHCR14, p. 698]. A reverse transformation from EPC graphs to EPClets is mentioned as useful but not described [KHCR14, pp. 698–699]. The DSL writes control flow as structured statements, whereas EPClets links rules through shared events. It targets executable BPMN and reads existing models back into source.

Drave et al. [DMM+22] built a textual BPMN notation to generate process-aware information systems from textual models. Its repository calls it the MontiCore Workflow DSL [MontiCore26].
````

It covers private, executable processes [DMM+22, p. 5]. Each gateway is written as a `split` or `merge` of a given kind, either declared by name or inline in a chain of sequence flows joined by arrows [DMM+22, p. 6]. Blocks group branches, and the node before a block decides how they run, for example all of them after a parallel gateway and one after an exclusive gateway [DMM+22, p. 6]. The node after a block decides how the branches join [DMM+22, p. 6]. Listing 1 of the paper writes a loop as a flow back to a declared merge gateway, and a task can also carry a looping behavior of its own [DMM+22, p. 6, Listing 1]. The tooling exports models as BPMN 2.0 XML, which the generated systems execute on an embedded Camunda engine [DMM+22, pp. 8–9]. The authors add that the tooling also transforms XML into the textual notation, but they do not describe this import [DMM+22, p. 16]. Laying out exported models remains manual work [DMM+22, p. 16]. The current public repository provides a command-line tool that parses, prints, and checks textual models, but neither direction of the XML transformation [MontiCore26]. According to its commit history, an XML export existed until March 2024, and an import into the textual notation never did [MontiCore26]. This thesis describes and evaluates its import of existing executable models (Sections 4.5.2 and 6.3).

Other front ends produce BPMN from program code or from a general text notation. The model application programming interface (API) of Camunda 7, its Java library for reading and writing BPMN models, contains a fluent builder API [Cam26b], which Operaton inherits [Ope26a]. Its call chains, such as `startEvent().userTask().endEvent()`, construct a model that can be serialized as XML [Cam26b]. It generates DI shapes and edges, but without auto-layout, so that "the elements of different branches may overlap," and its documentation calls it "not nearly complete" [Cam26b]. The builder can extend a model read from XML, but the documentation describes no path from an existing XML model back to builder code [Cam26b]. Of the works in this chapter, it is the only one that ships with the engine this thesis targets.

Loadshift is an internal Kotlin DSL for workflows with tasks, conditions, loops, parallel branches, fan-out over child items, timers, messages, and user tasks [Nel26]. It compiles each workflow through the Camunda 7 model API into BPMN with engine extensions [Nel26]. The output targets three platforms: Camunda 7, CIB seven, which is a fork of Camunda 7 [CIB26], and Camunda 8 [Nel26]. In our test, its output also runs on Operaton 2.1.0 (Section 3.11). Its conditions remain Kotlin functions: a generated service task evaluates each one, and the gateway after it only tests the stored result, so the compiled models need a running Loadshift worker [Nel26]. No path leads from XML back to Kotlin [Nel26].

Ivanchikj et al. [ISP20, pp. 238–239] report PlantBPMN, a textual DSL for BPMN from a master's thesis, built with Xtext and edited in Eclipse, as the only earlier work on textual modeling for BPMN known to them. Its tooling transforms between the metamodel of that DSL and the BPMN XML Schema, and the author writes constructs such as the kind of a gateway or of an event explicitly [ISP20, p. 238]. The DSL of this thesis derives its gateways from structured statements (Section 4.3).

A second project named PlantBPMN, unrelated to the first and listed as the PlantUML converter in Table 3.2, converts PlantUML activity diagrams into BPMN 2.0 XML for Flowable, an open-source BPMN engine [PlantBPMN26]. It supports actions and if-then-else blocks, but

no loops or parallel branches, and bracketed attributes set engine properties, such as the Java class of a service task [PlantBPMN26]. The tool writes the XML from templates and has no importer [PlantBPMN26]. The DSL has loops, parallel branches, and events as well, and it reads models back into source.

Miragon's bpmn-to-code [Miragon26] works in the opposite direction. It generates type-safe Java and Kotlin constants from BPMN models, such as element ids, message names, service-task types, and timer definitions, for Camunda 8, Camunda 7, and Operaton. It also offers a library for architecture tests over those models, rule checks that report, for example, missing implementations or undefined timers [Miragon26]. Its constants describe the model and do not compile back into one. The importer translates the control flow into source that compiles back into a model, and the equivalence relation of Section 4.5.4 compares this model with the original.

3.5 Zeebe's YAML Format

Camunda once shipped a textual front end for its process engine Zeebe, its own YAML format for simple workflows [Cam21b]. A YAML workflow was a list of tasks, executed top-down, with `goto`, `end`, and `switch` clauses for jumps and data-based branching [Cam20b]. On deployment, the engine translated it into a BPMN model of service tasks and exclusive gateways [Cam20a]. The format was deprecated with version 0.26.0 and removed with version 1.0.0-alpha1 [Cam21b; Cam21a]. One of the five reasons that Camunda gave for the removal is that "a text-based format only" is "inherently limited to workflows that are quite simple" [Cam21a]. This reason is a direct objection to a textual syntax for BPMN. However, the format it describes expressed only service tasks, jumps, and data-based branching [Cam20b; Cam20a]. It had no events beyond the implicit start and end, no parallel branches, and no sub-processes, and it translated in one direction only [Cam20b; Cam20a]. This thesis answers the objection with a larger element scope, nearly all of the BPMN that its engine executes (Section 4.2). The DSL compiles to BPMN 2.0 XML and reads it back, so the XML stays the format that engines and modeling tools exchange. On import, the DSL refuses constructs outside its scope or drops them with a warning (Section 4.2.1).

3.6 BPMN Sketch Miner

The BPMN Sketch Miner [ISP20; ISP22] is a web-based tool that generates BPMN diagrams from a constrained natural-language notation [ISP20, pp. 229–230]. Each text describes execution traces, and process mining infers the control flow from them, so that the diagram updates while participants describe the process [ISP20, pp. 229–231]. The tool exports standard BPMN 2.0 XML [ISP20, p. 232], but its models are sketches that are, in its authors' words, non-executable [ISP20, p. 230]. In their outlook, the authors mention two-way synchronization, including from the diagram back to the text, as a direction for their future work [ISP22, p. 1903]. In the DSL, the author writes the control flow directly as structured statements, and the resulting models are executable and can be read back into text.

3.7 kUML

kUML [Bet26b] is a statically typed internal DSL in Kotlin for several modeling notations, among them UML 2.x, SysML 2, C4, and BPMN 2.0, each with its own metamodel. Its BPMN support was added in version 0.16.0 and, at the cited version, covers process, collaboration, choreography, and conversation diagrams [Bet26b]. Authoring stays at the granularity of BPMN elements: each gateway is created, typically bound to a variable, and each sequence flow is connected explicitly [Bet26a; Bet26c] (Listing 3.5). kUML ships a language server and simulates process diagrams with tokens, but it does not deploy them to an engine [Bet26b]. It reads and writes BPMN 2.0 XML, and for process and collaboration models an export followed by an import preserves ids, names, and element types [Bet26b]. The importer yields an in-memory model and ignores unknown elements [Bet26b]. Its only printer of BPMN models as Kotlin source is internal to its transformation from UML activities, so an imported model cannot be printed back as source [Bet26b]. Its implementation has no support for extension elements, so it drops those of Camunda [Bet26b]. Its builders, Kotlin functions such as `task` and `gateway` that each create one model element, record only the kind of an event and the type of a task, so a kUML model holds no timer expression, no message reference, and no script [Bet26a].

Listing 3.5 writes the invoice approval of Section 3.11 in kUML in 573 non-whitespace characters, against 246 in the DSL. The listing creates each element with a builder, stores it in a variable, and then connects these variables with sequence flows [Bet26a; Bet26c]. One of the two flows out of the first gateway is marked as the default flow, which runs when the condition does not hold [Bet26a]. A task takes only a name, a task type, and loop or boundary markers, so the listing contains neither the assignees nor the delegate class, and the start event cannot ask for the amount [Bet26c], as the start form of the DSL does (Listing 3.6). kUML can also run a process in a simulation that moves a token through the model, and in this simulation an amount of 1500 reaches `SeniorApproval` and an amount of 500 reaches `AutoApprove`. Operaton 2.1.0 rejects the XML that kUML exports, since kUML writes the default mark as an attribute of the sequence flow, where the BPMN schema does not permit it. After we moved this mark to the gateway by hand, the file deployed, but Operaton created no runnable process from it, since kUML does not mark the process as executable.

The DSL is an external language with its own grammar and language server. It derives gateways and flows from conditionals, loops, and parallel blocks, so the author writes none of them (Section 4.3). The DSL targets execution on Operaton: it compiles and imports engine extensions, and its output is tested on the engine.

3.8 HyLiMo and Renew

HyLiMo [KSB24] is a hybrid text-and-diagram editor for the diagrams of technical publications, implemented as a Visual Studio Code extension on the LSP [KSB24, p. 70]. Its textual DSL, internal to the custom language SyncScript, is the single source of truth, and graphical edits

List. 3.5: The invoice approval in the BPMN DSL of kUML [Bet26a; Bet26c], as a script for its command-line tool, which imports the builders by default [Bet26b].

```
bpmnModel("Invoice Approval") {
    process(id = "invoice-approval", name = "Invoice Approval") {
        val start = startEvent("ReviewStart")
        val review = task("ReviewInvoice", type = TaskType.USER)
        val split = gateway(GatewayType.EXCLUSIVE)
        val senior = task("SeniorApproval", type = TaskType.USER)
        val auto = task("AutoApprove", type = TaskType.SERVICE)
        val join = gateway(GatewayType.EXCLUSIVE)

        start flowsTo review flowsTo split
        sequenceFlow(split, senior, condition = "amount > 1000")
        sequenceFlow(split, auto, default = true)
        senior flowsTo join
        auto flowsTo join flowsTo endEvent("Done")
    }
    diagram("Invoice Approval", processId = "invoice-approval")
}
```

are written back into the text [KSB24, pp. 71–72]. Its authors observe that other hybrid editors keep layout and style outside the text, whereas HyLiMo stores both in the text [KSB24, p. 70]. Renew [KWD+04], an editor and simulator for reference nets, a kind of object-oriented Petri net [KWD+04, p. 484], developed at the University of Hamburg, annotates the graphical Petri net with textual inscriptions, written primarily as Java expressions [KWD+04, p. 486]. Renew keeps the graph as the primary notation and writes only its inscriptions as text. Ciccozzi et al. [CTVW19, p. 425] define blended modeling as “the activity of interacting seamlessly with a single model (i.e., abstract syntax) through multiple notations (i.e., concrete syntaxes), allowing a certain degree of temporary inconsistencies.” Among the dimensions they use to classify such tools are the number of abstract syntaxes and whether changes propagate between notations sequentially or concurrently [CTVW19, pp. 428–429]. In these dimensions, the DSL keeps its text and its BPMN XML as separate artifacts and propagates changes sequentially, through an explicit compile or import, so it is not a blended modeling tool in this sense. HyLiMo and Renew serve diagrams outside process execution, whereas the DSL targets executable process models in an interchange standard and regenerates their layout.

3.9 LLM-Driven BPMN Generation

A separate line of research generates BPMN from natural-language text, either by combining natural language processing tools [FMP11, p. 482] or, more recently, with LLM-based tools, five of which Drakopoulos et al. [DMN+26, p. 3] compare. The approach of Friedrich et al. [FMP11, p. 483] places no restrictions on how its input text is structured. Such approaches translate unrestricted text into BPMN and offer no notation that an author writes and reads back,

so they do not compete with the DSL. Section 7.4.1 returns to them as possible users of the compiler diagnostics of the DSL.

3.10 Restructuring and Bidirectional Transformation

The importer of the DSL turns a gateway graph back into structured statements (Section 4.5.2), a problem that also arises between other pairs of process languages. An import into a notation that writes every gateway, such as the Workflow DSL, can map the graph element by element. A structured syntax cannot do this, and a round trip between block-structured and graph-based languages typically changes the model (Section 2.3). Mazanek and Hanus [MH11, p. 68] avoid this change by covering structured process models only. They write a bidirectional transformation between BPMN and BPEL as a graph parser in the functional logic language Curry. A context-free hypergraph grammar defines the models it covers [MH11, pp. 70–71]. These models consist of start and end events, activities, timer and message events, and parallel and exclusive blocks of two branches, each closed by its join [MH11, pp. 67–68, 71, 78]. Parsing such a model yields a BPEL term [MH11, pp. 79–80]. The parser is a functional logic program, so the same definition also runs backwards and constructs a BPMN graph for a BPEL term [MH11, p. 83]. The two directions therefore agree by construction.

In the DSL, the compiler and the importer are two separate programs, so their agreement does not follow by construction. In return, the importer also accepts unstructured graphs. It folds the gateway patterns it recognizes into blocks and prints a `goto` for each remaining edge (Section 4.5.2). Every model of their grammar can be written in the DSL with `step` (a plain task), `await` (a wait for an event), `if`, and `parallel` statements. Each of these statements corresponds to one constructor of their BPEL term [MH11, p. 79]: `step` to `Invoke`, `await` to `Wait` or `Receive`, `parallel` to `Flow`, and `if` to `Switch`. Their example process [MH11, p. 68, Fig. 2], written in this way, compiles, and import prints the same statements back. The elements of their models carry only labels, event kinds, and branch conditions [MH11, p. 78], so these models lack the settings that Operaton needs to run a process.

Hauser and Koehler [HK04] address the same restructuring problem for another pair of languages. They compile UML 2.0 activity diagrams, which allow concurrency and arbitrary cycles, into a simplified subset of BPEL that allows cycles only as `while` loops [HK04, p. 318]. For a sequential part of the graph, they first generate a program with `goto` statements and then remove the jumps with a goto elimination method from compiler construction [HK04, p. 323]. For an irreducible graph, that is, one with jumps from outside into the middle of a loop, the elimination has to copy code [HK04, pp. 319, 329]. It also introduces auxiliary variables, which must be meaningful if people are to read the program [HK04, p. 329]. Alternatively, they translate the graph into a single loop whose body selects the next node through a variable, so that every jump becomes an assignment to this variable [HK04, pp. 324–325]. They mention, without elaborating, that the two methods can be combined, for example by eliminating jumps as far as possible without copying code and translating the rest into such a loop [HK04, p. 329]. Their decompilation assumes generated code whose structure was left unchanged [HK04, p. 332]. Where their approach needs

duplicated code or such a loop [HK04, pp. 329–330], the DSL emits a `goto`. Its compiler accepts any valid source, whether generated or edited by hand.

Polyvyanyy et al. [PGD12, pp. 526–530] structure a model through its RPST (Section 2.3): they traverse the tree bottom-up and replace every rigid fragment that has an equivalent structured form by that form [PGD12, pp. 519, 521, Fig. 1]. Their method assumes sound models [PGD12, p. 519], in which every run can complete and then leaves no other token behind [PGD12, p. 522]. It is complete for acyclic models, handles cycles only where they form single-entry single-exit fragments, and excludes inclusive joins inside a rigid fragment [PGD12, pp. 536–537]. On 604 SAP reference models, Choi et al. [CHKH20, pp. 623–624, Table II] find 0.36 rigid fragments per model on average and at most two. In these models, a printer that confines `goto` to rigid fragments (Section 7.4.6) would print a `goto` in at most two fragments per model. The printer of the DSL needs none of these conditions, since it prints a `goto` for every edge it cannot fold into a block (Section 4.5.2).

Czarnecki et al. [CFH+09, pp. 261–262] describe a bidirectional transformation as a pair of one-way transformations, one per direction. In practice, they observe, the two are mostly written as separate programs, whereas a language that derives both from one definition can guarantee their compatibility by construction [CFH+09, p. 262]. The compiler and the importer of the DSL are such a pair, and round trips check their compatibility. Foster et al. [FGM+07, p. 1] design a language in which every expression denotes such a bidirectional transformation, called a lens. A lens pairs a get function, which maps a concrete view to its abstract view, with a putback function, which takes an edited abstract view and the old concrete view and returns an updated concrete view [FGM+07, p. 6, Def. 3.1]. Two round-trip laws state the expected compatibility: putting back an unchanged view returns the original concrete view (GetPut), and the view of a put-back result is the view that was put (PutGet) [FGM+07, pp. 6–7, Def. 3.2]. Foster et al. [FPP08, p. 383] relax these laws to hold only up to an equivalence relation chosen by the programmer, since realistic transformations change unimportant details such as whitespace. The equivalence relation of Section 4.5.4 takes this role for the DSL. With the source text as the concrete side and the XML as the abstract side, the compiler plays the role of get and the importer that of putback. The importer ignores any earlier source and regenerates the text from the XML. It is therefore the putback of an oblivious lens, whose putback gives the same result for every concrete view it receives [FGM+07, p. 9, Def. 3.7]. Synchronization re-establishes the relation between artifacts that exist in parallel [CFH+09, p. 271]. The importer does this in one direction only: when both artifacts were edited, it keeps the edits to the XML and discards those to the earlier source text.

3.11 One Process in Four Notations

Listings 3.6 to 3.9 write the invoice approval of Figure 2.1 in the DSL and in three related languages for executable BPMN: the Workflow DSL of Drave et al. [DMM+22], the fluent builder, and Loadshift. Each listing is checked with the tooling of its language. The engine tests of BPMNscript (Section 5.8) execute the process of Listing 3.6 on Operaton 2.1.0 with one amount above 1000 and one below. The Workflow DSL model passes the checker of its repository [MontiCore26].

The model built with the fluent builder deploys on Operaton 2.1.0 and routes both amounts to the intended task. The Loadshift workflow compiles and runs on Operaton 2.1.0 through the Loadshift worker. Sizes are given in non-whitespace characters, the measure that RQ3 uses (Section 1.3).

List. 3.6: The invoice approval in the DSL, the `invoice-approval` example process of BPMNscript [Kra26b] without its comment line.

```
process invoice-approval {
  start ReviewStart {
    form {
      amount: number "Invoice amount"
    }
  }
  user ReviewInvoice(assignee: "demo")
  if (amount > 1000) {
    user SeniorApproval(assignee: "manager")
  } else {
    service AutoApprove(class: "com.example.invoice.AutoApproveDelegate")
  }
  end Done
}
```

Listing 3.6 takes 246 characters. It writes no gateway and no sequence flow, and it states the form field of the start event, both assignees, and the delegate class that Operaton needs to run the process (Section 4.3.5).

The Workflow DSL (Listing 3.7) takes 332 characters and writes both gateways, the split and the merge, with the branches between them [MontiCore26]. Its grammar has no attribute for the assignee of a user task or for the Java class of a service task. Tasks are grouped by role only through lanes [DMM+22, p. 6], and the tooling ignores free-form stereotypes [MontiCore26]. The checker rejects a user or service task without the attribute `webservice`, which holds the BPMN implementation technology, so every such task sets it, here to `##unspecified` [MontiCore26]. Of the start form, only the variable remains, as an output of the start event, and the checker does not check this output [MontiCore26]. At the cited commit, the repository has no export to XML (Section 3.4), so the model cannot run on an engine with the current tooling.

The fluent builder (Listing 3.8) takes 879 characters and also writes both gateways. It reaches the second branch only by navigating back to the split, here with `moveToNode`, and leads that branch into the join with `connectTo` [Cam26b]. It expresses everything the DSL source states, but it needs statements outside the chain to set the default flow. Its only builder method for a default flow, `defaultFlow`, expects an existing flow object [Ope26a]. Inside the chain, that object is reachable only through a cast and a model lookup [Ope26a]. The listing therefore marks the default flow on the finished model, in three further statements. The listing also writes out the process and task names, which the compiler of the DSL derives from the identifiers (Section 5.4.2). It sets the history time to live to the 30 days that the compiler sets by default, since the builder would otherwise set 180 days [Ope26a].

List. 3.7: The invoice approval in the Workflow DSL of Drave et al. [DMM+22], in the syntax of its repository at the cited commit [MontiCore26]. The repository syntax differs from Listing 1 of the paper in keyword order, such as `user task` for `task user` [DMM+22, p. 6].

```
process InvoiceApproval {
  data amount : long;

  start event ReviewStart {
    out amount;
  }
  user task ReviewInvoice {
    webservice = ##unspecified;
  }
  user task SeniorApproval {
    webservice = ##unspecified;
  }
  service task AutoApprove {
    webservice = ##unspecified;
  }
  end event Done;

  ReviewStart -> ReviewInvoice -> split xor -> {
    [amount > 1000] SeniorApproval;
    [_] AutoApprove;
  } -> merge xor -> Done;
}
```

Loadshift (Listing 3.9) takes 328 characters and is the only one of the three that writes the choice as a conditional statement without a gateway, as the DSL does [Nel26]. Its condition, however, stays a Kotlin function, so `amount > 1000` does not appear in the model (Section 3.4). Every service task is an external task, which the engine leaves to an outside worker [Ope26a]: the Loadshift worker fetches it over the Representational State Transfer (REST) API and carries it out [Nel26]. A Java delegate class therefore cannot be bound [Nel26]. Loadshift also adds an external service task after each user task [Nel26]. It has no start form, and it gives the start and end events fixed identifiers [Nel26].

Of the three, the Workflow DSL and the fluent builder write every gateway, Loadshift moves the condition out of the model, and only the fluent builder produces a model that Operaton runs without further tooling.

Table 3.1 compares the four notations on single constructs: a sequence, a choice, a loop, a parallel split, and a jump back. The fragments of the fluent builder and of Loadshift run on Operaton 2.1.0 for both values of the condition, and those of the Workflow DSL pass the checker of its repository [MontiCore26]. The fluent builder has no loop construct, so it writes each loop as a gateway with a flow back to an earlier node [Cam26b]. The Workflow DSL writes its loop as a loop marker on a single task [DMM+22, p. 6] and needs two declared gateways for a jump back, since a merge gateway may not also split [MontiCore26]. Loadshift has no jump, so its jump back is a loop whose condition holds on the first pass [Nel26]. Loadshift and the DSL write no gateway, and only the DSL writes the jump back as a jump.

List. 3.8: The invoice approval with the fluent builder of the Operaton 2.1.0 model API [Cam26b; Ope26a].

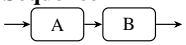
```
BpmnModelInstance model = Bpmn.createExecutableProcess("invoice-approval")
    .name("Invoice Approval")
    .operatonHistoryTimeToLiveString("P30D")
  .startEvent("ReviewStart")
    .operatonFormField()
      .operatonId("amount")
      .operatonLabel("Invoice amount")
      .operatonType("long")
    .operatonFormFieldDone()
  .userTask("ReviewInvoice")
    .name("Review Invoice")
    .operatonAssignee("demo")
  .exclusiveGateway("split")
    .condition("amount > 1000", "${amount > 1000}")
    .userTask("SeniorApproval")
      .name("Senior Approval")
      .operatonAssignee("manager")
    .exclusiveGateway("join")
    .endEvent("Done")
  .moveToNode("split")
    .sequenceFlowId("toAutoApprove")
    .serviceTask("AutoApprove")
      .name("Auto Approve")
      .operatonClass("com.example.invoice.AutoApproveDelegate")
    .connectTo("join")
  .done();
ExclusiveGateway split = model.getModelElementById("split");
SequenceFlow toAutoApprove = model.getModelElementById("toAutoApprove");
split.setDefault(toAutoApprove);
```

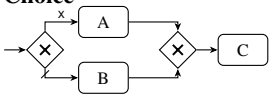
List. 3.9: The invoice approval in Loadshift [Nel26]. The imports, the list of invoices, and the function `autoApprove`, plain Kotlin defined elsewhere, are omitted.

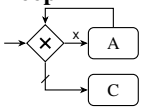
```
@Serializable
data class Invoice(val id: String, val amount: Double) : WorkItem {
    override val key get() = id
}

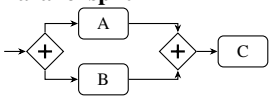
val invoiceApproval = workflow<Invoice>("invoice-approval") {
    input(invoices)
    userTask("ReviewInvoice", assignee = "demo")
    condition({ it.amount > 1000 }) {
        userTask("SeniorApproval", assignee = "manager")
    } otherwise {
        task("AutoApprove") { autoApprove(it) }
    }
}
```

Table 3.1: Five constructs in the BPMN notation and in four textual notations, checked as in Section 3.11. `A`, `B`, and `C` stand for tasks, `step` is the plain task of the DSL, `x` for a Boolean variable, and `a`, `b`, and `c` for Kotlin functions. Declarations, process frames, and start and end events are omitted, except the events `S` and `E` of the Workflow DSL.

BPMN	Workflow DSL [MontiCore26]	Fluent builder [Cam26b]	Loadshift [Nel26]	DSL
Sequence 	`S -> A -> B -> E;`	`.userTask("A")` `.userTask("B")`	`task("A") { a(it) }` `task("B") { b(it) }`	`step A` `step B`

Choice

	`S -> split xor -> {`
	`  [x] A;`
	`  [_] B;`
	`} -> merge xor -> C -> E;`
		`.exclusiveGateway("split")`
		`  .condition("x", "${x}")`
		`  .userTask("A")`
		`.exclusiveGateway("join")`
		`.moveToNode("split")`
		`  .condition("not x", "${!x}")`
		`  .userTask("B")`
		`  .connectTo("join")`
		`.userTask("C")`
			`condition({ it.x }) {`
			`  task("A") { a(it) }`
			`} otherwise {`
			`  task("B") { b(it) }`
			`}`
			`task("C") { c(it) }`
				`if (x) { step A }`
				`else { step B }`
				`step C`

Loop

	`task A while [x];`
	`...`
	`S -> A -> C -> E;`
		`.exclusiveGateway("test")`
		`  .condition("x", "${x}")`
		`  .userTask("A")`
		`  .connectTo("test")`
		`  .condition("not x", "${!x}")`
		`  .userTask("C")`
			`loop({ it.x }) {`
			`  task("A") { a(it) }`
			`}`
			`task("C") { c(it) }`
				`while (x) { step A }`
				`step C`

Parallel split

	`S -> split and -> {`
	`  A;`
	`  B;`
	`} -> merge and -> C -> E;`
		`.parallelGateway("fork")`
		`  .userTask("A")`
		`.parallelGateway("join")`
		`.moveToNode("fork")`
		`  .userTask("B")`
		`  .connectTo("join")`
		`.userTask("C")`
			`parallel {`
			`  branch { task("A") { a(it) } }`
			`  branch { task("B") { b(it) } }`
			`}`
			`task("C") { c(it) }`
				`parallel {`
				`  { step A }`
				`  { step B }`
				`}`
				`step C`

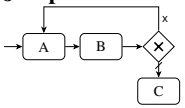
Jump back

	`merge xor Again;`
	`split xor Repeat;`
	`...`
	`S -> Again -> A -> B -> Repeat`
	`-> {`
	`  [x] Again;`
	`  [_] C -> E;`
	`};`
		`.userTask("A")`
		`.userTask("B")`
		`.exclusiveGateway("test")`
		`  .condition("x", "${x}")`
		`  .connectTo("A")`
		`.moveToNode("test")`
		`  .condition("not x", "${!x}")`
		`  .userTask("C")`
			`loop({ it.first || it.x }) {`
			`  task("A") { a(it) }`
			`  task("B") {`
			`  it.first = false; b(it) }`
			`}`
			`task("C") { c(it) }`
				`step A`
				`step B`
				`if (x) { goto A }`
				`step C`

Table 3.2: Related textual process languages compared, grouped as in this chapter. The last three columns rate requirements R1 (control flow as program statements, gateways and sequence flows derived), R2 (compiles to executable BPMN 2.0 XML), and R3 (imports executable XML) of Section 1.1: ✓ holds, ∼ holds partially, ✗ does not hold, ? reported without description, – does not apply, which includes R2 and R3 for a work that targets a language other than BPMN. Ext. = external DSL with its own grammar; Int. = internal DSL in a host language.

Work	DSL kind; editor	Control flow	Executes on	BPMN XML	Diagram layout	R1	R2	R3
This thesis	Ext. (Langium); LSP, VS Code	block, `goto` fallback	Operaton	writes and reads	generated (library)	✓	✓	✓
BPELscript [BKLL09]	Ext. (ANTLR); in progress	block, links	any BPEL engine	no; reads and writes BPEL	–	∼	–	–
StalkCD [DKH21]	YAML of a metamodel; not reported	nested stages	Jenkins; BPMN not executable	writes and reads	generated (dagre)	✓	✗	✗
WEE / CPEE [MR22; MSS10]	Int. (Ruby); web editor	block	own engine	accepts as input	own editor	✓	✗	✗
HiProX [MSS10; VPMR25]	Int. (Rust), generated; via CPEE	block	compiled instances	via CPEE	via CPEE	✓	✗	✗
TOSCA, GitHub Actions [GH26; LC25]	YAML format; not surveyed	step or job graph	TOSCA orchestrators; GitHub	no	–	✗	–	–
EPClets [KHCR14]	Ext. (Xtext); Eclipse	event rules; connectors constructed	none (EPC)	no; reverse not described	generated	∼	–	–
Workflow DSL [DMM+22; MontiCore26]	Ext. (MontiCore); command line	gateways written; blocks group branches	Camunda, embedded	writes; reading stated, not described	manual	✗	∼	?
Fluent builder [Cam26b]	Int. (Java); host language	graph	Camunda 7, Operaton	writes	generated, without auto-layout	✗	✓	✗
Zeebe YAML [Cam20b; Cam21a]	YAML format; not reported	list, `goto`, `switch`	Zeebe (removed in 1.0)	compiled to a model	not reported	✓	∼[1]	✗
Loadshift [Nel26]	Int. (Kotlin); host language	block; conditions as Kotlin code	Camunda 7 and 8, CIB seven, Operaton; workers evaluate conditions	writes	generated	✓	✗	✗
PlantBPMN [ISP20]	Ext. (Xtext); Eclipse	constructs written explicitly	not reported	transformation reported	not reported	✗	?	?

[1] The engine translated the YAML into a BPMN model on deployment [Cam20a], so the format ran only on Zeebe.

Table 3.2: Related textual process languages compared (continued).

Work	DSL kind; editor	Control flow	Executes on	BPMN XML	Diagram layout	R1	R2	R3
PlantUML converter [PlantBPMN26]	PlantUML subset; not reported	if/else blocks	Flowable	writes	not reported	∼	✓	✗
Sketch Miner [ISP20]	constrained text; web editor	mined from traces	none	writes	generated	∼[2]	✗	✗
kUML [Bet26b]	Int. (Kotlin); LSP	graph	token simulation	writes; reads into a model, not source	not reported	✗	✗	✗
HyLiMo [KSB24]	Int. (SyncScript); LSP, VS Code	diagrams only	none	no; text and diagram kept in sync	stored in the text	–	–	–
Renew [KWD+04]	Java inscriptions; own editor	Petri net	own simulator; workflow engine via plug-in	no	own editor	–	–	–

[2]The author writes execution traces as ordered lists of tasks, and process mining derives the gateways and flows from them [ISP20, p. 230]. Parallel tasks are declared on one line, separated by | [ISP20, pp. 231–232].

3.12 Gap Analysis and Positioning

The answer to RQ1 is that textual process languages relate text and graphics in four ways, and a few works fall outside them (Table 3.2). Some replace the graphical model with code and give up a standardized graphical notation, such as the workflow-as-code platforms (Section 2.7), TOSCA, and GitHub Actions. Some keep their own engine and language and import graphical models, such as CPEE, with HiProX as its compiled variant. Some generate BPMN from text in one direction, such as the Sketch Miner, the fluent builder, Zeebe's former YAML format, Loadshift, and the PlantUML converter. Few translate in both directions: BPELscript for BPEL, StalkCD for CI/CD pipelines, kUML for engine-neutral diagrams, with an import that yields an in-memory model, and the MontiCore Workflow DSL, whose import from BPMN XML its authors state without describing it. Outside these four, TN4PM and EPClets serve modeling or translation between notations, bpmn-to-code generates constants from models, HyLiMo and Renew combine text and diagrams outside BPMN, and the generators from natural language offer no notation to write.

None of these works meets the three requirements stated in Section 1.1. The MontiCore Workflow DSL comes closest [DMM+22], since it is the only one that reports both an export to executable XML and an import from it. It writes every gateway, so it misses R1, and its authors state an import from XML without describing it (Section 3.4). They report that their tooling exports BPMN 2.0 XML that an embedded Camunda engine runs [DMM+22, pp. 8–9]. In our test with the tooling of January 2022 [MontiCore22], whose grammar follows Listing 1 of the paper, the export of the invoice approval drops the branch condition and the data object and writes no diagram layout. Operaton 2.1.0 rejects the result: a flow out of the gateway has neither a condition nor the default mark, the service task has no implementation, and the process has no history time to live. We could not reproduce an executable export, so Table 3.2 marks R2 as partially met. Two other works come close on one requirement. kUML imports executable XML into an in-memory model [Bet26b]. Its import misses R3, since kUML cannot print an imported model as source and drops the Camunda extensions, timer expressions, message references, and scripts (Section 3.7). Loadshift writes control flow as blocks, and in our test its output runs on Operaton (Section 3.11). Its conditions stay Kotlin functions, so its models route only with a running Loadshift worker and miss R2 (Section 3.4). For the other works of Table 3.2, the table records which of the three requirements each misses.

Chapter 4 designs a language that meets the three requirements, starting from the element scope that Operaton defines (RQ2).

4 Approach

This chapter designs the DSL and its transformation to and from BPMN 2.0 XML, meeting the three requirements stated in Section 1.1. Section 4.1 first states the design objectives and non-goals. Section 4.2 answers RQ2 (Section 1.3) by deriving the constructs the DSL needs to cover from Operaton's deployment parser. Sections 4.3 to 4.5 answer the design part of RQ3 with the concrete syntax, the intermediate representation and both directions of the XML transformation. The chapter closes with the equivalence relation and its tolerances, by which RQ4 judges a round trip (Section 4.5.4). Chapter 5 realizes this design, and Chapter 6 evaluates it.

All design decisions behind this chapter are recorded as ADRs in the Markdown Architectural Decision Records (MADR) format [KAZ18, p. 57]. The 33 ADRs are published with release 0.1.1 of BPMNscript [Kra26b], the release this thesis describes, and Appendix H lists them with the section each decision belongs to. This chapter discusses the decisions with the largest effect on the language, the transformation, and the scope, each with its reason and the options set aside, and leaves the smaller ones to their records. A reference of the form ADR-*nnnn* in parentheses links to the full record of the decision.

4.1 Design Objectives and Non-Goals

BPMNscript delivers a language server for editors, a compiler, and an importer, and the language server and the compiler share one validator (Figure 4.1).

Five objectives guide the design of the DSL. Developer orientation and conciseness refine R1, engine executability refines R2, and bidirectionality refines R3 (Section 1.1), and IDE support serves the developers whom R1 addresses. The DSL is *developer-oriented*: it assumes general programming literacy but no prior knowledge of BPMN. Its keywords are derived from common programming terms for the underlying action instead of the BPMN element that represents it (ADR-0006). For this audience, the DSL departs from the advice that a DSL "should be as close as possible to the notation used in a domain" [MHS05, p. 331]. For process modelers, that notation is the graphical notation of BPMN. It is *concise*: whatever the compiler can infer, such as gateways, sequence flows, and identifiers, need not be written in the source. It is *engine-executable*: every construct the DSL exposes compiles to BPMN 2.0 XML that Operaton deploys and runs. The validator reports an error wherever the source breaks one of a set of the engine's parse-time rules, so the engine accepts every compiled model under these rules. It is *bidirectional*: existing BPMN artifacts can be imported into the DSL, and a process written in the DSL can be compiled to BPMN 2.0 XML, edited in any other modeling tool, and imported again. This exchange concerns

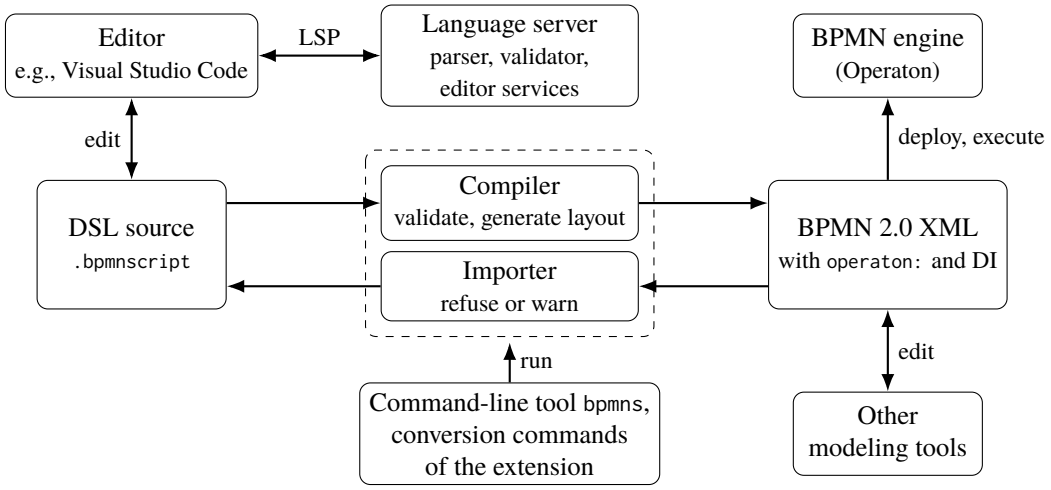


Figure 4.1: The tools that BPMNscript delivers. An editor talks to the language server over the LSP and receives diagnostics, completion, hover, navigation, and rename. The compiler turns the source into BPMN 2.0 XML and refuses source with validator errors, and the importer turns such XML back into source. The command-line tool and the conversion commands of the Visual Studio Code extension run both.

the process model, which a round trip has to preserve up to the equivalence relation (Section 4.5.4). Lastly, it is *IDE-supported*: a language server provides diagnostics, completion, hover, navigation, and rename over the LSP (Section 5.3).

Five non-goals bound the scope of the design. Diagram layout is not authored: the DSL carries no coordinates, and the DI data of BPMN is regenerated on compilation and discarded on import (Section 4.5.3). Portability across BPMN engines is out of scope, since each engine fills what the specification leaves open with its own choices (Section 2.4). Collaborations are not authored: a DSL file holds only one process (Section 4.2.1). Choreography and conversation diagrams are excluded as well, because no engine executes them (Section 2.1): the message exchanges of a conversation "will be executed ultimately through an orchestration Process" [OMG13, Sect. 9.5, p. 125]. Finally, round trips are not byte-identical. What the transformation preserves is an equivalence relation (Section 4.5.4).

4.2 Scoping the Executable BPMN Subset

The scope of the DSL is driven by which constructs the targeted BPMN engine, Operaton (Section 2.5), supports. A construct counts as supported if the engine deploys and executes it. Deployment alone does not suffice: the engine keeps some deployed elements without runtime effect, and a statement for such an element changes nothing at runtime. Section 4.2.1 derives this set of executed constructs, the *executable subset*, element by element. Two kinds of content without runtime effect are added to it, because they carry information for the developer. Documentation travels with the exported model, and variable declarations let the validator check each use of a

variable (Section 4.2.4). The conformance sub-classes of the specification are considered as an alternative scope (Section 4.2.2), and usage studies of public models set the priority within the scope (Section 4.2.3).

4.2.1 The Executable Subset of Operaton

The executable subset is read off the deployment parser of Operaton 2.1.0 [Ope26a], which decides at deployment what each element becomes. For every element the parser does one of four things:

(1) It gives the element a runtime effect.
(2) It reads or tolerates the element without any runtime effect.
(3) It fails the deployment.
(4) It never reads the element at all.

Only the first outcome puts an element into the executable subset, and every classification comes from reading the parser source.

For events, the outcome depends on two properties: the type of the event, such as message or timer, and its position in the process, for example the start of a process or the boundary of a task. A catching event waits until something of its type happens, and a throwing event sends something of its type. Operaton distinguishes eleven types: the none event, which waits for or sends nothing in particular, and ten others such as message, timer, and error. In the model, an event of one of the ten types carries an *event definition* that specifies its type. Together with ten positions, this gives 110 combinations (Appendix A). The specification permits only 54 of those combinations [OMG13, Tables 10.84–10.86, 10.88–10.90, 10.92, Sect. 10.6.6], and Operaton executes all 54 [Ope26a]. Of the 56 combinations the specification does not permit, Operaton deploys four as written and 17 as none events, ignoring their type, and fails the deployment for 35 [Ope26a]. For example, Operaton deploys and runs a non-interrupting error boundary event, although an error event always interrupts the activity it is attached to [OMG13, Table 10.90, p. 254]. It fails the deployment of a boundary event without a type, and it deploys a timer end event but ignores its timer, so the event runs as a none end event [Ope26a]. Import refuses all 56 of them (Appendix A).

All other elements are counted by element type. An element type is a class of the BPMN metamodel [OMG13], such as `userTask` or `lane`. Without the five event classes, the count gives 36 element types (Appendix B). Operaton executes 19 of them [Ope26a], listed in Table 4.1. The association takes effect in one role only: it links a compensation boundary event to the activity that performs the compensation [Ope26a]. A BPMN `property` element also takes effect, on a user task only: Operaton turns it into a variable local to the task at runtime [Ope26a]. It counts as part of the user task (Appendix B), and import drops it with a warning. Operaton reads four element types without runtime effect: the pool, the lane, the documentation, and the DI data. It ignores the ad-hoc sub-process and the complex gateway with a warning, and fails the deployment when a sequence flow touches either. It does not parse the other eleven. Table 4.1 summarizes these

counts. Appendix B lists every element type with its import behavior and the DSL construct that Section 4.3 introduces for it.

For each construct, import compares the behavior of the engine with what the DSL can write. A construct the engine rejects is refused. What the engine runs and the DSL can express is carried into a DSL construct. A construct the engine accepts but the DSL cannot express is dropped with a warning. It is refused instead where dropping it would change what runs, as for a call activity that starts a case of the Case Management Model and Notation (CMMN) (Section 4.2.4). Engine settings that the intermediate representation (IR) does not model, such as an execution listener on a sequence flow, are an exception. The IR carries engine settings as typed fields, which the validator can check, so a setting without a field cannot be carried (ADR-0021), and import drops it with a warning (Section 4.5.4). Two simpler rules are considered (ADR-0012). Refusing every construct the DSL cannot express drops nothing the engine runs, but it refuses a file over content such as a pool or a lane, which Operaton reads only for the diagram (Table 4.1). Dropping such content without a warning keeps these files importable, but the developer then has to compare the imported source with the original to find what the rebuilt process lacks.

A file with more than one executable process is refused, although Operaton deploys each of them as a definition of its own [Ope26a], because the IR (Section 4.4) and a DSL file hold one process. In the corpus of the evaluation, it refuses 144 of the 6,060 models that Operaton deploys (2.4 %) for holding several executable processes (Section 6.6). Lifting this bound, or splitting such a file on import into one source file per process, is future work (Section 7.4.3).

Import repairs two constructs that the engine does not deploy, and warns about each repair. A process without a history time to live, the retention period of its history, receives the default (Section 4.3.5). A lone process that is not marked executable is skipped by the engine [Ope26a], so import marks it executable. Import refuses an error, escalation, or compensation start event on the process, although the engine runs it as a plain start [Ope26a], because the specification does not permit it there [OMG13, Table 10.84]. Section 4.5.4 lists what import drops, with and without a warning.

4.2.2 Conformance Sub-Classes as a Scope

The sub-classes of Process Modeling Conformance (Section 2.1) offer a ready-made alternative scope, but two reasons speak against them. First, the Descriptive and Analytic sub-classes grade what a modeler draws. They include pools, lanes, data objects, and the standard loop marker [OMG13, Tables 2.1, 2.2], which Operaton keeps only for the diagram or does not parse at all (Table 4.1). Second, the Common Executable sub-class, the one sub-class concerned with execution, does not describe the behavior of Operaton either. It omits the script task, the business rule task, the inclusive gateway, timer start and timer boundary events, event sub-processes, and every signal, escalation, and compensation event [OMG13, Tables 2.3, 2.4], all of which Operaton executes. It also requires three standard languages, which Operaton does not use: XML Schema for data types, the WSDL for service interfaces, and XPath for data access [OMG13, Sect. 2.2.2]. In contrast, Operaton binds services through its own extension attributes (Section 2.5) and evaluates

Table 4.1: Element types by their outcome in Operaton 2.1.0 (Section 2.5), with their import into the DSL (Section 4.3). Events are counted by combination (Appendix A), and Table B.1 lists each element type.

Operaton	Types	Element types	Import into the DSL
executed	19	process; task, user, service, send, receive, script, business rule, and manual task; sub-process, transaction, call activity; multi-instance marker; exclusive, parallel, inclusive, and event-based gateway; sequence flow; association	17 carried into a construct; manual task rewritten into a `step`; association to a compensation handler refused
read without runtime effect	4	pool, lane, documentation, diagram interchange	documentation carried into a setting; pool and lane dropped with a warning; diagram regenerated
ignored with a warning; a flow to or from it fails the deployment	2	ad-hoc sub-process, complex gateway	refused
not parsed	11	standard loop marker, message flow, text annotation, group, data object, data object reference, data store reference, data input, data output, data input association, data output association	dropped with a warning
total	36		

expressions in the Java Unified Expression Language (JUEL) [Ope26a]. The conformance type BPEL Process Execution Conformance (Section 2.1) adds support for the mapping of BPMN to BPEL [OMG13, Sect. 2.4], which Operaton does not need, since it executes BPMN directly. Since the DSL targets Operaton, neither the sub-classes nor the conformance types are a suitable scope for it, and the executable subset is derived from the engine's actual behavior instead.

4.2.3 Empirical Secondary Justification

Usage studies do not decide the scope, but show what it needs to cover in practice. Compagnucci et al. [CCFR24, p. 47] collected 39,695 distinct public BPMN 2.0 models from seven collections, the largest of them from the BPM Academic Initiative and from GitHub. Only 543 of them are choreographies and 289 are conversations, so excluding both (Section 4.1) affects about 2 % of the models. They analyzed the remaining 38,863 process and collaboration models. Table 3a of Compagnucci et al. [CCFR24, p. 51] lists 21 elements that occur in more than 10 % of the models, counting the exclusive gateway with and without its X marker as one. Of these 21 elements, 13 lie inside the executable subset, and the other eight, such as the expanded pool, the lane, and the data object, carry no execution semantics in Operaton (Table 4.1). However, frequency does not limit the scope, since in specific domains every element of the notation may find its use [CCFR24, p. 65], so the scope includes every element Operaton executes. Frequency does set the priority within the scope instead. The syntax is designed around the frequent constructs, so that they take the least text: a task is one statement, an exclusive gateway is an `if`, and sequence flows as well as

the start and end events of a body follow from the order of the statements (Section 4.3). Rarer constructs fit around them and may take more text.

4.2.4 Coverage and Exclusions

Data objects stay out of the DSL, because Operaton does not parse them (Table 4.1). The engine works on process variables instead, which it creates when a form, a mapping, or the start of the process first sets them [Ope26a]. The DSL lets the author declare them as `var name: type`. The validator uses these declarations to check each use of a variable in a condition or a mapping, and to check the declared type against the operator. The declarations are not compiled to BPMN 2.0 XML, because BPMN has no element for them. Import therefore cannot recover a declaration. The printer, the part of import that writes DSL source (Section 4.5.2), gives the type `any` to each variable the process reads, unless a form field, a handler, an input or output parameter, or a `for` clause declares it (Section 4.3.4). Every variable that the imported source reads is thus declared.

This answers RQ2. Operaton executes 19 element types and all 54 event combinations that the specification permits (Section 4.2.1). The DSL has a construct for 17 of the 19 element types. The first exception is the manual task, a task performed without the aid of an engine or an application [OMG13, Sect. 10.3.3]. Since Operaton runs it like a plain task [Ope26a], import rewrites it into a `step`, the DSL statement for a plain task (Table 4.3), and warns the developer about this engine behavior. The second exception is the association. Its one role at runtime is to link a compensation boundary event to its handler [Ope26a]. The DSL writes such a handler as an `on compensation` block, for the reason given below for the compensation boundary event. Import therefore refuses this association together with its boundary event. It drops any other association, such as one to a text annotation, with a warning. The call activity has a construct, but import refuses one variant of it, a call activity that starts a case instead of a process. A case is a model in the CMMN, a separate standard, which Operaton runs in an engine of its own [Ope26a]. CMMN describes work without a fixed course: case workers decide at runtime which tasks a case needs and in which order to perform them [OMG16, Sect. 4.1, p. 6]. Since the DSL covers BPMN only, `call` refers to a process, and import refuses a call activity that refers to a case. Dropping the reference instead would leave the call activity without a target.

Of the four element types that Operaton reads without runtime effect, the DSL has a construct for the documentation only. The pool and the lane stay out, because Operaton reads them for the diagram only and the DSL writes no collaboration (Sections 4.1 and 7.4.3), and the compiler generates the layout (Section 4.5.3). Import refuses the ad-hoc sub-process and the complex gateway, since a sequence flow touching either fails the deployment. It drops the eleven element types that Operaton does not parse with a warning, which leaves the running process unchanged.

Of the 54 event combinations, the DSL carries 52. The first exception is the none intermediate throw event, which marks a point in the flow without waiting for or sending anything. Operaton passes the token through it [Ope26a], so import rewrites it into a `step` with a warning, as it does for the manual task. The second exception is the compensation boundary event. Compensation undoes a completed activity, and BPMN offers two places for the handler that performs the

undo [OMG13, Sect. 10.7.1]. The first is a single activity outside every sequence flow, attached by an association to a compensation boundary event of the completed activity [OMG13, Sect. 10.7.1, p. 302]. The second is an event sub-process inside the sub-process to be undone. The DSL supports only the second, since the body of a DSL handler is a sequence of statements joined by sequence flows. It writes this event sub-process as an `on compensation` block, a handler without a host that is placed inside the sub-process it undoes (Section 4.3.3). Import therefore refuses the boundary form, and its error message shows the equivalent block.

4.3 DSL Syntax Design

This section presents the concrete syntax of the DSL. It introduces the running example (Section 4.3.1) and chooses the shape of an element from six candidate syntaxes (Section 4.3.2). The grammar, the naming and scoping of identifiers, and the settings that Operaton reads from its own namespace follow in Sections 4.3.3 to 4.3.5.

4.3.1 Running Example

The expense claim of Listing 1.1 on page 4 is the running example of this chapter, a process constructed for this thesis and published with the examples of BPMNscript [Kra26b]. Section 4.5.1 shows the BPMN XML that the compiler generates for it. In Section 4.5.2, we import a variant of it as a modeler might draw it, with flows that are not nested in blocks (Section 2.3). The compiled process with its generated layout is shown in Figure 1.2, and its XML is listed in Appendix C.

The two variable declarations (lines 2–3) let the validator check that `amount` and `compliant` are used with their declared types. The start event `ClaimSubmitted` carries a form (lines 5–9), which Operaton offers to the user who starts the process to set the `amount`. The service task `CheckPolicy` (line 11) calls a Java class, a delegate, that checks the claim against the expense policy and sets `compliant`. The `while` loop (lines 13–20) repeats the user task `ReviseClaim` as long as the claim does not comply, and the task's form sets both variables again. The `if` statement (lines 22–26) routes large claims to a manager, and the `parallel` block (lines 28–35) books the payment and notifies the claimant concurrently, with each branch in its own pair of braces. The compiler derives three exclusive gateways, two parallel gateways, and fifteen sequence flows from the order and nesting of the statements (Section 4.5.1).

4.3.2 Design Rationale

Every process, activity, and event of the DSL is written in one shape:

```
keyword Id(settings) { contents }
```

The parentheses hold the scalar settings that describe the element, such as its label. The braces hold everything with internal structure: the flow inside a process or sub-process and the members of an activity, such as form fields, listeners, and input and output mappings (Section 4.3.5). An element without members has no braces, and an element without settings has no parentheses. An element that carries an event states its trigger, which is the type of the event, such as timer or error. The trigger's payload, for example the duration of a timer, comes first in the parentheses: `keyword Id trigger(payload, settings)`. A control statement such as `if` has no name, and its condition in parentheses takes the place of the name, so every statement of a process body opens with its keyword.

Listing 4.1 shows this shape in the `order-handling` example process. It contains a user task with two settings, a sub-process with three statements, and two handlers. Each handler is attached to its host activity with `on Host:` and reacts to a timer or an error while the host runs. The error handler binds the code and the message of the caught error to the variables `c` and `m`. The timer handler's payload `"PT4H"` is a duration of four hours in the ISO 8601 notation, which BPMN requires for the durations of timers [OMG13, Table 10.101], and `alongside` lets the host continue when the timer fires instead of interrupting it.

List. 4.1: Excerpt of the `order-handling` example published with BPMNscript [Kra26b], in the chosen syntax. Three further handlers are omitted.

```
process order-handling {
  error PAYMENT_DECLINED(message: "The card could not be charged")
  escalation LARGE_PAYMENT_FLAGGED

  start OrderPlaced
  user ReviewOrder(label: "Review order", assignee: "demo")
  subprocess Payment(label: "Process payment") {
    service AuthorizePayment(label: "Authorize payment", class: "com.example.demo.LogDelegate")
    emit escalation(LARGE_PAYMENT_FLAGGED)
    service CapturePayment(label: "Capture payment", class: "com.example.demo.LogDelegate")
  }
  service ShipOrder(label: "Ship order", class: "com.example.demo.LogDelegate")
  end OrderShipped

  on ReviewOrder: timer("PT4H", alongside) {
    service SendReviewReminder(label: "Send a review reminder", class: "com.example.demo.LogDelegate")
  }
  on Payment: error(PAYMENT_DECLINED, code: c, message: m) {
    user ContactCustomer(label: "Contact the customer about the decline", assignee: "demo")
  }
}
```

This shape is chosen from six candidate syntaxes (ADR-0011): colon configuration (A), parentheses and braces (B), keyword prose (C), the off-side rule (D), decorator annotations (E), and the binding form (F). Each candidate is drafted as a complete rendering of an order-handling process that is larger than Listing 4.1 (Appendix D). Table 4.2 summarizes all six, with excerpts of variants B, D, and F in Listings 4.2 to 4.4. The drafts spell handlers, timer durations, and a few

settings differently from the final syntax of Listing 4.1, as in `on Pack error OUT_OF_STOCK(…)` and `timer(after: "PT2H")`, and the criteria below do not depend on this spelling. The candidates are compared by four criteria in decreasing weight: how they mark settings and contents, whether every statement opens with a keyword, the cost of implementing them, and their familiarity to the target audience. Since the renderings are our own and no user study compares them, the comparison is analytical and shows that B is defensible on these criteria, but not that developers prefer it.

List. 4.2: A user task, a sub-process with one child, and an error handler in variant B, the chosen syntax (excerpt of Listing D.2).

```
user Review(label: "Review the order", assignee: "demo")
subprocess Pack(label: "Pack the order") {
  service Weigh(label: "Weigh parcel", class: "com.example.WeighDelegate")
}
on Pack error OUT_OF_STOCK(code: c, message: m) {
  user Contact(label: "Contact customer", assignee: "demo")
}
```

List. 4.3: A user task, a sub-process with one child, and an error handler in variant D, the off-side rule (excerpt of Listing D.4).

```
user Review:
  label: "Review the order"
  assignee: "demo"
subprocess Pack:
  label: "Pack the order"
  service Weigh:
    label: "Weigh parcel"
    class: "com.example.WeighDelegate"
on Pack error OUT_OF_STOCK:
  catch: code c, message m
  user Contact:
    label: "Contact customer"
    assignee: "demo"
```

Settings and contents. Under variant B (Listing D.2), the delimiter after an element's name says whether settings or contents follow, just as the BPMN XML serialization separates an element's attributes from its child elements. Variants A (Listing D.1) and D (Listing D.4) mark both the same way, with braces or with indentation, so the reader has to tell from the first word of each line whether it is a setting or a nested statement. Under variant B, one task also shows the shape of every activity and event, since every setting except the trigger payload and a flag such as `alongside` is written `key: value` inside the parentheses (Section 4.3.3). Variant C (Listing D.3) instead writes details as prose with connective words, such as `labeled` and `catching`, whereas B writes them as keyed settings, whose keys the validator checks and the editor offers for completion. Variant E

List. 4.4: A user task, a sub-process with one child, and an error handler in variant F, the binding form (excerpt of Listing D.6).

```
Review = user(label = "Review the order", assignee = "demo")
Pack = subprocess(label = "Pack the order") {
  Weigh = service(label = "Weigh parcel", class = "com.example.WeighDelegate")
}
on Pack error OUT_OF_STOCK (code = c, message = m) {
  Contact = user(label = "Contact customer", assignee = "demo")
}
```

(Listing D.5) writes the settings of an element as annotation lines above its keyword, so the reader meets the settings of an element before its keyword and has to read it from the bottom up.

Size. Size, the compactness that Karsai et al. [KKP+09, Guideline 20] ask to balance against comprehensibility, cannot separate the candidates. By characters, the size measure of RQ3, the six renderings lie within 15 % of each other, and variant C, which writes its settings as prose, is the shortest (Table 4.2).

Table 4.2: The six candidate syntaxes: how each marks an element's settings and its contents, and the size of the order-handling process in each, as non-whitespace characters and as lines, blank lines included (Appendix D).

Variant	Settings	Contents (flow, members)	Characters	Lines
A, colon configuration	`{ key: value }`	`{ }`	1,207	71
B, parentheses and braces (chosen)	`( key: value )`	`{ }`	1,201	53
C, keyword prose	connective words	`{ }`	1,079	53
D, off-side rule (indentation)	indented `key: value`	indentation	1,171	79
E, decorator annotations	`@key value` lines above	`{ }`	1,156	88
F, binding form	`Id = kind(key = value)`	`{ }`	1,231	51

Keyword-first statements. Variant F (Listing D.6) is the strongest alternative: it marks settings and contents with different delimiters, as B does, and is about as long. It is rejected mainly because of how its statements open. F writes every element as a binding of a name, `Review = user(...)`. Control statements and handlers declare no name, so F's own rendering (Listing D.6) writes `if` and `on` as keyword statements. A statement of F thus opens either with a name the author invents or with a keyword, whereas every statement of B opens with its keyword. Voelter [Voe13, p. 139] advises the designer of a textual language to consider how far the IDE can support a given syntax, above all through code completion. Variants F and B differ in this respect. In B, completion can offer every statement kind at the start of a line. In F, it can offer the element kinds only after a name and an equals sign. The binding also suggests that a name holds a value that can be reassigned, although the language has no reassignment. F's own rendering shows the problem, since it binds `amount` twice, as a variable and as a form field (lines 2 and 8 of Listing D.6). Karsai

et al. [KKP+09, Guideline 21] ask that "the same look-and-feel should be used for all sublanguages and especially for the elements within a language," which B meets with one keyword-first shape for elements and control flow alike.

Implementation cost. Variant D (Listing D.4) is the one candidate whose rejection rests mainly on the implementation. It expresses nesting through indentation and follows the off-side rule of Landin [Lan66, p. 160], under which a phrase, except for bracketed parts, must lie within the area to the right of and below its first symbol. Adams [Ada13, p. 511] notes that "context-free grammars cannot express the rules of indentation," so that parsers for such languages fall back on ad hoc techniques. Langium, the language workbench the DSL is built with (Section 5.2), supports such languages since version 3.2 with a token builder that inserts synthetic tokens wherever the indentation grows or shrinks [Langium26g]. For the DSL, this mechanism has three costs. First, the grammar file no longer describes the language completely, since the token builder decides part of its structure outside the grammar rules. Second, block boundaries are no longer characters the user typed. A missing block boundary is a missing synthetic token, so the editor first translates that parse error into terms of the source and then reports it at the right line. Worse, a change of whitespace alone can form a different valid program, whereas under variant B the nesting changes only where a brace is moved, which the reader sees. Third, the printer would have to compute the indentation of every nested line, and an indentation defect in the printer would change the structure of the printed process instead of its appearance. Karsai et al. [KKP+09, Guideline 24] state the general rule, with indentation as their example: "Prefer layout which does not affect translation from concrete to abstract syntax."

Familiarity to the target audience. The shape `name(arguments) { body }` resembles the method declarations of Java, Groovy, and Kotlin. The delegate classes that a process binds (Section 4.3.5) are written in one of these languages, and Operaton itself is written in Java [Ope26a]. The audience of Section 4.1 can therefore be assumed to know the shape. Where no suitable notation exists in a domain, Karsai et al. [KKP+09, Guideline 14] advise adopting the new language "as close as possible to other existing notations within the domain or to other common used languages." Since BPMN has no textual notation meant for authoring (Section 1.1), and its graphical notation does not suit the audience of the DSL (Section 4.1), this fallback applies. Familiarity helps only users who already program in the existing language [MHS05, p. 326]. For novices who had never programmed, Stefik and Siebert [SS13, pp. 19:1, 19:26] found that languages with a traditional C-style syntax (Perl and Java) "did not afford accuracy rates significantly higher than a language with randomly generated keywords." In their survey, however, programmers rated the syntax of C++ higher the more years of C++ experience they reported, by roughly half a point for each year on a rating scale of eleven points [SS13, pp. 19:21–19:22]. The target audience, especially the developers who write the delegate classes, can be assumed to know such a language, so familiarity supports the choice of variant B.

Result. Variant B is therefore chosen for its separate delimiters and uniform keyed settings, for the keyword-first shape that F lacks, and for avoiding the costs that D would put on the editor support and the printer.

4.3.3 Lexical and Grammatical Structure

A process opens with its name and optional settings, lists its declarations, and then its body, a sequence of statements that runs from top to bottom. Sequence flows are not written, since two consecutive statements are connected by a flow, and a body that leaves out its start or end statement receives a generated one (Section 4.5.1). A variable declaration, `var amount: number`, gives a variable one of the types `string`, `number`, `boolean`, `date`, `json`, and `any`. A code declaration, such as `error PAYMENT_DECLINED(message: "…")`, declares the code of an error or an escalation (Section 4.3.4).

The body consists of the twenty statement kinds of Table 4.3, and since each of them opens with its keyword, the parser never has to guess where one statement ends and the next begins. The settings of a statement stand in one pair of parentheses that holds at least one item. An item is a keyed setting `key: value`, a bare flag such as `alongside`, or one unkeyed value, the trigger payload, which can be a code, a message name, a condition, or a timer value. A script body is a fenced block, ```` ```groovy ... ``` ````, that the lexer reads as a single token, so that script text is never read as DSL source.

Each control statement *lowers* to the gateways and flows that express it (Section 4.5.1). An inclusive gateway activates every outgoing flow whose condition holds. Its join waits for every token that can still reach it, and in a block these are the tokens of the activated branches [OMG13, Sect. 13.4.3]. The DSL therefore writes it as a `parallel` block with conditioned branches. An event-based gateway, a race in which the first of several events to occur decides the branch [OMG13, Sect. 10.6.6], is an `await` block with one branch per event. Five constructs have no gateway form: boundary events, event sub-processes, compensation, transactions with their cancellation, and link events. The DSL therefore writes them with the `on`, `attempt`, `emit`, and `await` statements of Table 4.3 instead of a control statement. A boundary event, an event attached to the boundary of an activity that catches its trigger while the activity runs [OMG13, Sect. 10.5.4], is a handler attached to its host, `on Host: timer("PT4H") { … }`. An event sub-process, a sub-process that an event starts while its enclosing process or sub-process runs [OMG13, Sect. 10.3.5], is a handler without a host. Compensation is the `on compensation` block described in Section 4.2.4. A transaction is a sub-process whose cancellation ends its running activities and compensates its completed ones [OMG13, Sect. 10.5.1, p. 234]. The DSL writes it as an `attempt` block, its cancellation as a `cancel` end, and the reaction to the cancellation as a `cancel` handler. A pair of link events, a throw event that jumps to the catch event of the same name within one process level [OMG13, Sect. 10.5.5, pp. 266–267], is written as `emit link` and `await link`. The DSL keeps this pair for imported link events, so that they survive the round trip. Where import cannot fold an edge into a block, it prints a `goto` (Section 4.5.2). Figure 4.2 shows the BPMN fragments that the control statements and handlers compile to. For the constructs whose form is chosen among several

candidates, Table 4.4 sets the chosen form against the rejected ones, with the reason recorded in the decision.

Table 4.3: The twenty statement kinds of the DSL and the BPMN elements they compile to. `Id` stands for the name of the element, `trig` for a trigger, `c` for a condition, and `M`, `d`, and `p` for the name of a message, a decision, and a process.

Statement	Generated BPMN elements
`start Id trig(...)`	`startEvent`, optional trigger definition
`end Id terminate`	`endEvent`, optional terminate or cancel definition
`user Id(...)`	`userTask`
`service Id(class: ...)`	`serviceTask`
`script Id` with fenced body	`scriptTask` with `script`
`step Id`	`task`
`send Id(class: ...)`	`sendTask`
`receive Id(message: "M")`	`receiveTask`, `message` root
`decide Id(decision: "d")`	`businessRuleTask`
`if (c) {...} else {...}`	`exclusiveGateway` split and join
`while (c) {...}`	one `exclusiveGateway`, back-edge flow
`do {...} while (c)`	one `exclusiveGateway`, back-edge flow
`parallel { {...} {...} }`	`parallelGateway` pair; `inclusiveGateway` pair if conditioned
`goto Target`	`sequenceFlow` only
`subprocess` / `attempt Id {...}`	`subProcess` / `transaction`
`call Id(process: "p")`	`callActivity`
`on Host: trig {...}`, `on trig {...}`	`boundaryEvent`; event `subProcess`
`throw trig(...)`	`endEvent` with trigger definition
`emit trig(...)`	`intermediateThrowEvent` with trigger definition
`await trig(...)`, `await {...}`	`intermediateCatchEvent`; `eventBasedGateway` and exclusive join

Conditions are written in an expression sub-language, a subset of the JUEL expressions that Operaton evaluates. A condition is parsed into an AST, which lets the validator check each variable reference against its declaration and the editor offer hover information and jump-to-definition on it. Word operators such as `and` and `eq`, method calls, and functions lie outside the subset. An expression that needs them is written as a raw template, a quoted string opening with `${` or `#{`, which the compiler passes to the engine as written (ADR-0032).

The reserved words are exactly the 42 keywords of the grammar: the words that open a statement or a clause and a few fixed values. Every other word of the vocabulary, such as a trigger kind, a setting key, or a listener event, is a *soft keyword* [Python26, Sect. 2.3.2]: it is lexed as an identifier and checked in its position by the validator. A process may thus declare `var message: string`, although `message` is also a trigger kind. An identifier may contain inner hyphens, as in `invoice-approval`, but may not end in one.

4.3.4 Naming, References, Scoping

The author must name every activity and every start and end event, and may name a `throw`, `emit`, or `await` statement, in the *name slot* that the statement offers for it. The name becomes the BPMN

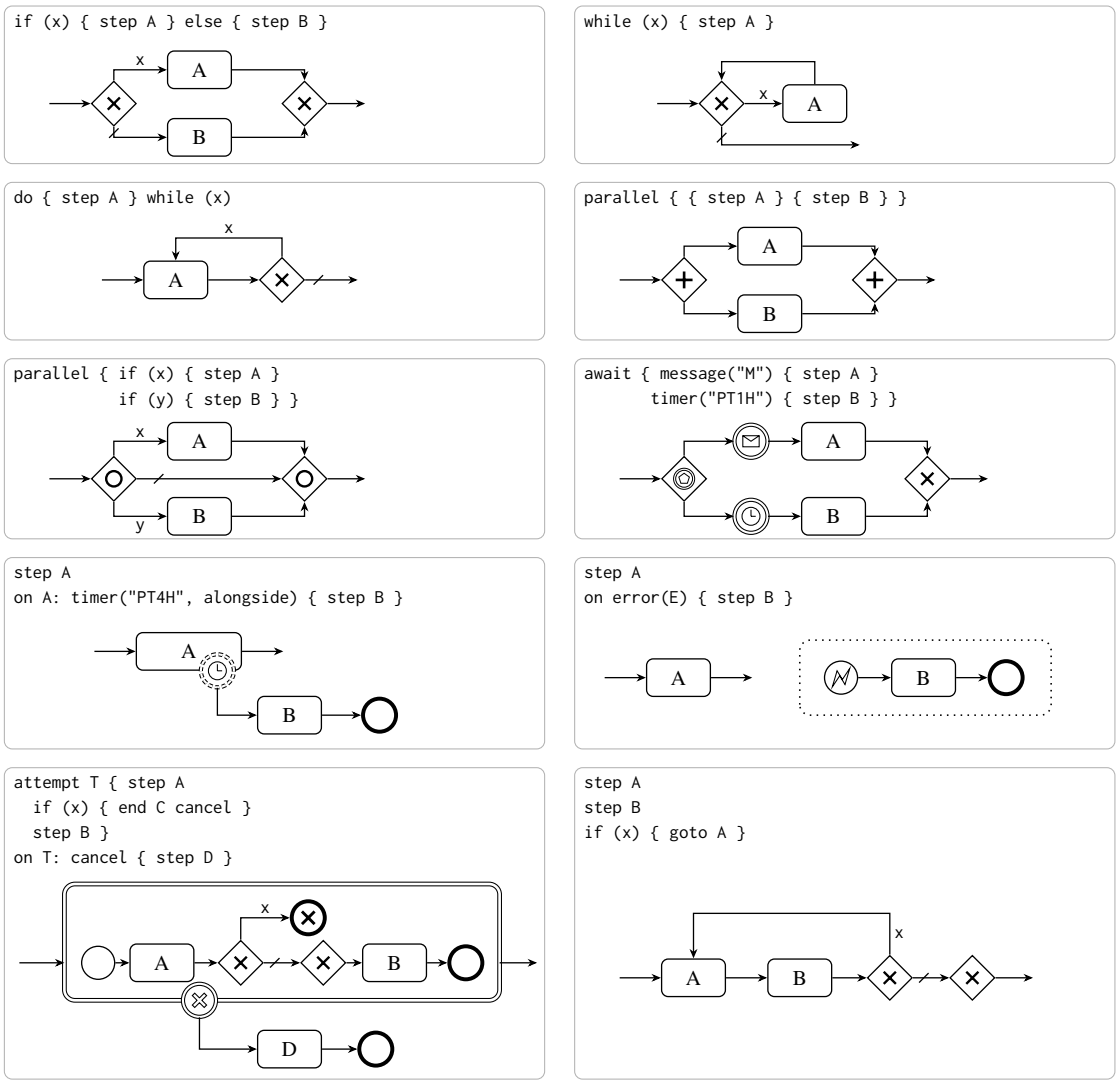


Figure 4.2: Statements of the DSL, each above the BPMN 2.0 fragment that release 0.1.1 compiles it to, where `step` is a plain task. Arrows at the edge of a fragment connect it to the surrounding statements, and the start and end events of the process are omitted. A slash marks the default flow of a gateway, and a dashed border marks a non-interrupting event. The compiler generates every start and end event shown, except the cancel end event `C`.

Table 4.4: Chosen and rejected forms of DSL constructs, with the reason recorded in the decision. `{...}` stands for a block of statements.

BPMN element	Chosen form	Rejected form	Reason
◇ exclusive gateway	`if (x) {...} else {...}`, `while (x) {...}`	gateways and tasks declared one by one and joined by `->` edges	the author draws the diagram in text, and a condition stays an unparsed string (ADR-0011)
⊕ parallel gateway	`parallel { {...} {...} }`	fork and join declared and joined by `->` edges	the author draws the diagram in text (ADR-0011)
⟶ sequence flow to an earlier element	`goto A`	`->` edges for every flow	a jump is written only where a flow has no block form (ADR-0011)
✉ catch event	`await message("M")`	`wait message("M")`	reads as a fixed pause (ADR-0026)
		`receive message("M")`	fits only a message (ADR-0026)
◈ event-based gateway	`await { message("M") {...} timer("PT1H") {...} }`	`race { ... }`	spends a word that authors may want as a name (ADR-0031)
		branches that rejoin by `goto`	forces a `goto` for the common timeout case (ADR-0031)
○ inclusive gateway	`parallel { if (x) {...} if (y) {...} }`	`parallel { (x) {...} (y) {...} }`	no other line of the language opens with a parenthesized expression (ADR-0031)
		a keyword of its own	spends a word on the name of the BPMN element (ADR-0031)
◇ exclusive split without default flow	`if (x) {...} else fail`	`else throw`	a thrown error can be caught, whereas Operaton stops such a split with an engine exception [Ope26a] (ADR-0031)
		`fail` as a reserved word	takes `var fail` and `user fail` out of the language (ADR-0031)
Ⓜ boundary event	`on A: error(E) {...}`	`on error(E) at A {...}`	`at` is already the key of a timer date (ADR-0027)
▭ transaction	`attempt T {...}`	`transaction T {...}`	promises atomicity, whereas Operaton runs it as a plain sub-process [Ope26a] (ADR-0029)
		`subprocess T(transaction: true)`	a Boolean setting that only renames the element (ADR-0029)
≡ multi-instance marker	`step A for each v in c`	`step A(each: c, as: "v")`	the keys repeat the BPMN attributes and admit `as` without `each` (ADR-0022)
		`for each v in c {...}`	adds a sub-process around an imported repeated task (ADR-0022)
		`for each` and `repeat 3` as two heads	refuses a document with both a count and a collection (ADR-0022)

identifier of the element, and the `label` setting becomes its displayed name. All other elements receive an identifier that the compiler synthesizes by the scheme of Section 4.5.1. These are the *elements without a name slot*, such as gateways, sequence flows, and boundary events, and the elements whose name slot stays empty, such as an unnamed `throw` statement or a start event that the compiler adds. The validator refuses an authored name that matches the pattern of a synthesized one, and the compiler resolves any remaining clash with a numeric suffix, so every generated identifier is unique.

A *flow container* is the process, a sub-process or `attempt` block, or a handler without a host, and corresponds to a flow elements container of BPMN [OMG13, Sect. 8.4.8, p. 86]. References are resolved in the flow container that holds them. The compiler looks up a `goto` target or the host of a handler among all statements of the enclosing flow container, at any block depth, and stops at nested flow containers, so a reference stays within its sub-process or event sub-process. The body of a handler with a host belongs to the flow container of its host. A jump from outside into one branch of a `parallel` or `await` block is refused, because the branch would then be entered without the gateway that opens it. Error and escalation codes match a thrown error or escalation to the handler that catches it [OMG13, Tables 8.41, 8.42]. Each code is declared once at the top of the process and referenced by its bare name wherever it is thrown or caught. A bare word resolves to a code only in a code position and only to a declaration of the matching kind, so the same word in a condition is read as a variable. A quoted code at every use site would need no declaration, but the language server would then have no declaration to jump to or rename, and a misspelled code would compile to a second error or escalation (ADR-0023). Messages and signals need no declaration, since the engine matches them by name only [Ope26a]. For reference resolution, variables are process-wide, even where the engine keeps a value local to an element, as for an input mapping. They are declared by `var`, by form fields, by the fields a handler binds, such as `code: c`, by input and output parameters, and by a `for` clause. A `for` clause is the multi-instance marker of BPMN: it runs an activity a given number of times or once per element of a collection, and it declares the element variable and the loop counters of the engine. The validator only warns about an undeclared variable, because a variable can also enter a process from outside, for instance when a process instance is started with variables.

A BPMN file written by another tool may contain identifiers that a plain DSL identifier cannot spell, such as `Task.1` or the keyword `user`. Every name slot therefore also takes a name in backticks, and the printer writes such an identifier in that form, as in `user \`Task.1\``, so that the document compiled from the imported source carries the same identifier. In contrast, the printed source has no place for an identifier that another tool gave an element without a name slot, so the compiled document carries a synthesized one instead, without a warning. Import refuses an identifier that Operaton rejects, such as one containing a colon, which the BPMN schema also excludes by typing every identifier as `xsd:ID` [OMG13, Table 8.17].

4.3.5 Operaton-Specific Constructs

A process that Operaton executes needs more than control flow: its service tasks must be bound to code, its user tasks may carry forms, and the engine may be told where to commit its database

transactions. Operaton reads these details from its own `operaton:` namespace (Section 2.5). The DSL writes all of them as settings or members in the shape of Section 4.3.2, under names that say what they mean, such as `retryCycle` for `failedJobRetryTimeCycle`, and the `operaton:` prefix is added only when the XML is written.

A service, send, or business rule task carries exactly one *binding*, the setting that identifies the code the engine runs for the task, and the validator refuses a task with no binding or with two. `class` binds the task to a Java delegate class, `expression` to a JUEL expression, `delegate` to an expression that resolves to a delegate object, `topic` to an external task fetched by a worker, and `type` to a built-in mail or shell behavior [Ope26a]. A `decide` task may instead be bound to a decision table with `decision`. A form is a `form` block in the braces of the process's start event or of a user task, with one field per line, as in Listing 1.1. Input and output mappings copy values into variables local to an element and back out of them, and listeners call code on events of the element's lifecycle, such as its start or end [Ope26a]. Both are members in the braces of their element, as are the variable mappings of a call activity. Field injection, written `field name = value`, hands name and value pairs to the object behind a `class`, `delegate`, or `type` binding [Ope26a]. Five job settings tell the engine where to end a database transaction and continue the instance in a background job, and how to retry that job. `asyncBefore` and `asyncAfter` set an *asynchronous continuation* before or after the element, and `exclusive`, `jobPriority`, and `retryCycle` configure the job.

The process header takes `versionTag`, `historyTimeToLive`, the candidate starter users and groups, who may start the process, and whether Operaton's task list offers the process for starting. The history time to live is the age after which the history cleanup of the engine may remove the records of a finished instance. In its default configuration, Operaton requires this value but sets no default for it, so it refuses a new deployment of a process without it [Ope26a]. The compiler therefore writes `P30D`, thirty days, when the source sets no value, and the printer leaves exactly that value out again.

The DSL contains no Java code. Embedding the body of a delegate would cut boilerplate further, but BPMN XML has no representation for Java source, so the body would be lost on compilation and could not be recovered on import. Delegates are therefore written in the application that embeds the engine, and a `class` setting is checked only when the engine runs the task. Comments serve only the author and are not compiled. Documentation for the exported model is a setting, `documentation: "…"`, which compiles to the BPMN `documentation` element.

4.4 The Intermediate Representation

A translator between two concrete formats can map one syntax directly onto the other or route both through a model of its own, independent of either notation. The DSL takes the second route: its IR is a statically typed graph of one process. A process holds flow elements of sixteen kinds and the sequence flows between them. The kinds are five events (start, end, intermediate throw, intermediate catch, and boundary), five tasks (plain, user, service, script, and receive, with send and business rule tasks as variants of the service task), the four gateways, the sub-process, and the

call activity. A sub-process is a nested flow container of the same shape, and its sequence flows stay inside it. A boundary event refers to its host by an attribute instead of a flow, and the two ends of a link pair refer to each other by name only. The IR has no blocks: an `if` or a `parallel` exists only in the source text, and in the IR it is a split gateway, a join gateway, and the flows between them. Every engine setting is an optional field of its node, under the unprefixed name of Section 4.3.5. The binding of a task is a single field with its kind and its target, such as a class name. Figure 4.3 shows the `if` statement of the running example in the IR.

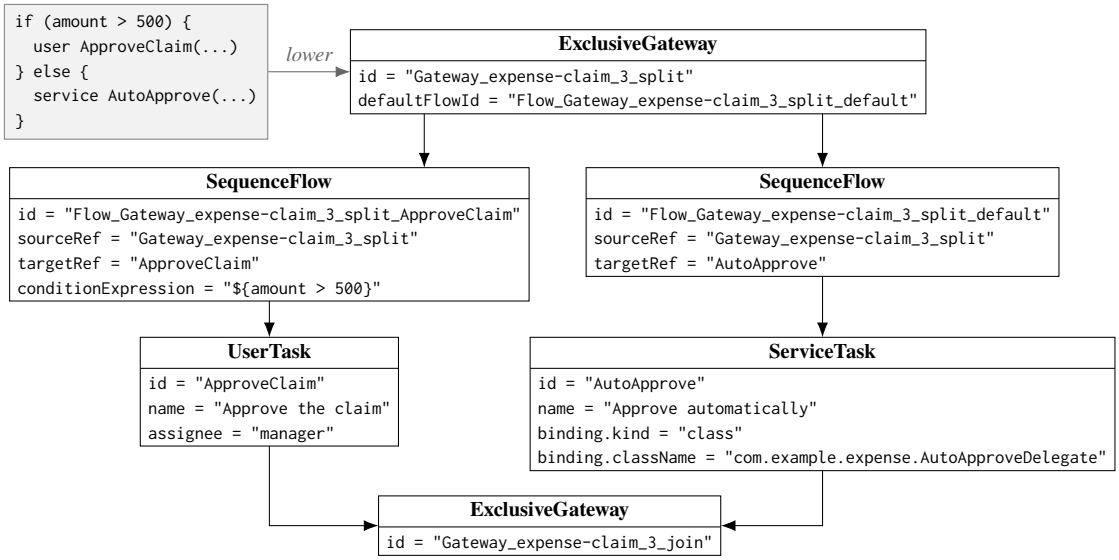


Figure 4.3: The `if` statement of the running example in the intermediate representation, with other fields omitted and the two flows into the join drawn as plain arrows. The statement becomes two exclusive gateways, two tasks, and the sequence flows between them, and the engine settings become plain fields of the tasks (Listing 4.5 shows the generated XML).

Three reasons favor one IR over a direct mapping from the syntax tree to XML (ADR-0007). First, the two central algorithms of the tool need a typed graph: the synthesis of gateways on compilation and the restructuring of the graph into blocks on import (Section 4.5). The syntax tree is not such a graph, and neither is the object model that a library builds from the XML. Second, both directions meet in one model, so a round trip is compared within a single representation, and the equivalence relation of Section 4.5.4 is stated on it. Third, a change of the grammar does not reach the serialization code, and namespaces, DI, and extension attributes do not reach the grammar. The cost is mapping code on both sides: every construct must be modeled in the IR and mapped to the syntax tree and to the XML.

On the DSL side, parsing followed by lowering builds the IR from source, and printing writes the IR back as source. On the XML side, serialization writes the IR as BPMN 2.0 XML, and reading builds the IR from such a document (Figure 4.4). The compiler is thus parsing, lowering, and serialization, and import is reading followed by printing. The *importer* performs import and has two parts: the *XML reader*, which performs reading, and the *printer*, which performs printing.

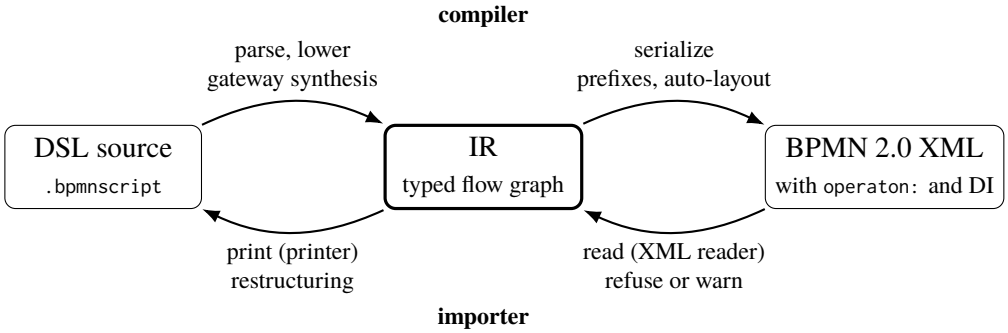


Figure 4.4: The four transforms of the DSL meet in one intermediate representation. The upper two form the compiler, and the lower two form the importer.

4.5 Bidirectional Transformation Strategy

The two directions of the transformation do different work. Compiling the DSL lowers nested blocks to a graph, whereas importing has to recover blocks from a graph that need not have a structured equivalent (Section 2.3). The two directions are described in Sections 4.5.1 and 4.5.2, followed by the handling of the diagram (Section 4.5.3) and the equivalence relation by which a round trip is judged (Section 4.5.4).

4.5.1 DSL to BPMN 2.0 XML

Structured statements lower to gateways. An `if`/`else if`/`else` chain becomes one exclusive split and one exclusive join. Each condition sits on the flow into its branch, and the `else` branch is the split's default flow. An `if` without `else` still receives a default flow, which runs to the join. The author can instead close a chain with `else fail`, which states that one of its conditions must hold. The split then has no default flow, so the engine stops the instance with an error when no condition holds [Ope26a]. A `while` loop becomes a single exclusive gateway in front of its body. A flow from the end of the body leads back to this gateway. Such a flow, which returns to an earlier element of the loop, is a *back edge*. The loop condition sits on the flow into the body, and the exit is the gateway's default flow. A `do` … `while` loop places the gateway after the body and puts the condition on the back edge. A `parallel` block becomes a parallel fork and join, or an inclusive pair when a branch is headed by a condition. A branch headed by `else` becomes the default flow of the inclusive fork. When every branch is conditioned, the default flow runs to the join, unless `else fail` closes the block as for `if`. A multi-branch `await` becomes an event-based gateway with one catch event per branch and an exclusive merge after them. A `goto` becomes a single sequence flow to its target. Two rules govern joins. If every branch ends or jumps away before the join, the join is removed, since it can never receive a token. A join that exactly one flow reaches stays, because Operaton deploys and runs it [Ope26a], although the specification requires every gateway to merge or split the flow [OMG13, p. 289]. Through such a join, the

printer recognizes on import an `if` whose branch ends or jumps away, such as a guard clause, a conditional early exit (Section 4.5.2).

Synthesized identifiers are deterministic (ADR-0010). A gateway, a catch or throw event, and an event sub-process take their identifier from the position of the statement that produces them. A sequence flow takes its identifier from its two ends. For example, the loop of the running example is the statement at index 2 of the body, counted from zero, so its gateway is `Gateway_expense-claim_2_loop`. Compiling unchanged source always yields the same identifiers. Listing 4.5 shows the elements that serialization generates for the `if` statement of the running example.

List. 4.5: The elements generated for `if (amount > 500)` of Listing 1.1. The two flows into the join and all `incoming` and `outgoing` children are omitted, and long attribute lists are wrapped.

```
<bpmn:exclusiveGateway id="Gateway_expense-claim_3_split"
    default="Flow_Gateway_expense-claim_3_split_default" />
<bpmn:exclusiveGateway id="Gateway_expense-claim_3_join" />
<bpmn:userTask id="ApproveClaim" name="Approve the claim"
    operaton:assignee="manager" />
<bpmn:serviceTask id="AutoApprove" name="Approve automatically"
    operaton:class="com.example.expense.AutoApproveDelegate" />
<bpmn:sequenceFlow id="Flow_Gateway_expense-claim_3_split_ApproveClaim"
    name="amount &#62; 500" sourceRef="Gateway_expense-claim_3_split"
    targetRef="ApproveClaim">
  <bpmn:conditionExpression xsi:type="bpmn:tFormalExpression"
    >${amount &gt; 500}</bpmn:conditionExpression>
</bpmn:sequenceFlow>
<bpmn:sequenceFlow id="Flow_Gateway_expense-claim_3_split_default"
    sourceRef="Gateway_expense-claim_3_split" targetRef="AutoApprove" />
```

Root elements are the elements directly below `definitions` [OMG13, Table 8.1], such as the errors and messages that events refer to. The compiler derives them from use: one `bpmn:error` or `bpmn:escalation` per distinct code, including a declared code that is neither thrown nor caught, and one message or signal per distinct name. A hosted handler becomes a boundary event, non-interrupting when marked `alongside`. Its body follows the boundary event in the host's flow container and ends in a generated end event unless it ends or jumps elsewhere itself. A handler without a host becomes an event sub-process with the trigger on its start event.

Start and end events are generated only for a whole body: the body of a flow container receives both, and the body of a hosted handler receives only an end event, since its boundary event is its entry.

4.5.2 BPMN 2.0 XML to DSL

Import reads a document into the IR and prints the IR as structured source. It follows the rules of Section 4.2.1, and a refusal aborts the whole import wherever it occurs, so a document is imported either completely or not at all. Printing then turns the flat graph into nested blocks. Constructs

without a block form, such as boundary events and event sub-processes, do not pass through this step. They are printed from their attachment, as `on Host:` and as a handler without a host (Section 4.3.3).

Retaining `goto`. Some graphs have a structured equivalent only with auxiliary variables, and some parallel ones have none at all (Section 2.3). Such a variable records which path the process has taken. The structured form then needs extra steps that set the variable and extra conditions that test it. The author never modeled these steps or the variable, so the rebuilt process differs from the one the engine runs, and the equivalence relation counts the gained nodes as a divergence (Section 4.5.4). The DSL therefore keeps `goto` in the grammar as the escape for edges without a block form in the catalogue below. Forcing every fragment into blocks also does not always help the reader. In an experiment by Dumas et al. [DLM+12, pp. 31, 44], an unstructured model was sometimes easier to interpret than its structured equivalent and sometimes harder, so structuredness "is not an absolute desideratum" for understandability.

The pattern catalogue. The printer analyzes the flow graph of each flow container with dominators and post-dominators (Section 2.3). The nearest post-dominator of a split, the first node that all of its branches reach again, is the natural candidate for its join. First, the printer leaves out every element unreachable from any start event, boundary event, or event handler, each with a warning, since Operaton never runs it [Ope26a]. It then applies a pattern catalogue with seven patterns. An exclusive split with its join prints as `if`, and two kinds of back edge print as `while` and `do … while`. A parallel or inclusive fork with its join prints as `parallel`, and an event-based gateway prints as `await`. An activity or event with several outgoing flows prints as the fork that the engine runs it as. Every other edge prints as `goto`. For an exclusive split, the printer also decides where its branches end and printing continues. Six rules, tried in a fixed order, decide this. The first two continue at the merge that pairs with the split as a closing bracket pairs with an opening one, or at the nearest post-dominator of the split, and the other four cover splits without such a join. Appendix F lists the passes, the patterns, and the rules with their exact conditions.

Figure 4.5 and Listing 4.6 show the catalogue on a graph-based variant of the running example, as a modeler might draw it. The variant adds a rejection after `ApproveClaim` and leaves out the forms and the parallel block. The back edge from `ReviseClaim` leads into a task, so neither loop pattern applies, and it prints as `goto CheckPolicy`. The compliance split joins again at the amount split, so it prints as an `if` with one branch. `ClaimRejected` ends a branch, so the amount split has no post-dominator inside the process. Its branches still meet at the merge in front of `BookPayment`, and printing continues there. Compiling the printed source yields six exclusive gateways instead of four. The two `if` statements with a single branch, one that jumps away and one that ends, each keep a join that only one flow reaches (Section 4.5.1). Tolerance (iii) of the equivalence relation removes such pass-through gateways (Section 4.5.4). Importing that compiled document again yields the same source.

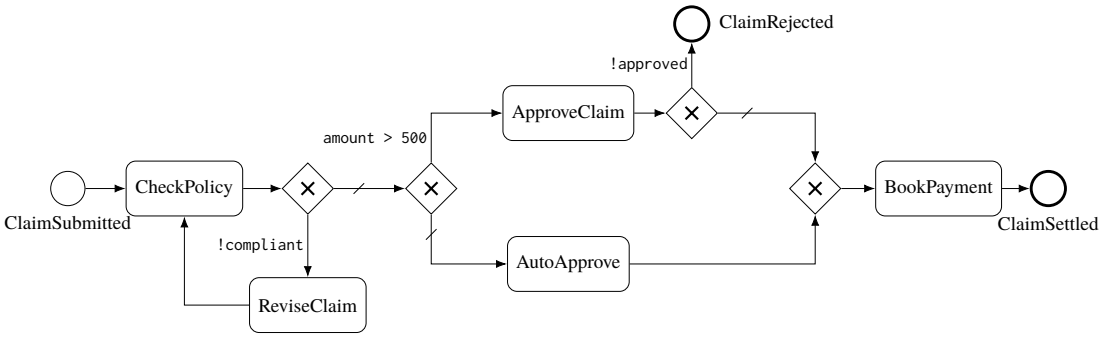


Figure 4.5: A graph-based variant of the running example (Listing 1.1), as a modeler might draw it. All four gateways are exclusive, a slash marks the default flow of a gateway, and the markers of the task types are omitted.

List. 4.6: The source printed for Figure 4.5, with one blank line added.

```
process expense-claim {
  var amount: any
  var approved: any
  var compliant: any

  start ClaimSubmitted
  service CheckPolicy(label: "Check the expense policy", class: "com.example.CheckPolicyDelegate")
  if (!compliant) {
    user ReviseClaim(label: "Revise the claim", assignee: "demo")
    goto CheckPolicy
  }
  if (amount > 500) {
    user ApproveClaim(label: "Approve the claim", assignee: "manager")
    if (!approved) {
      end ClaimRejected
    }
  } else {
    service AutoApprove(label: "Approve automatically", class: "com.example.AutoApproveDelegate")
  }
  service BookPayment(label: "Book the payment", class: "com.example.BookPaymentDelegate")
  end ClaimSettled
}
```

What the pattern catalogue does not fold. A split without a matching pattern prints as an `if` chain of jumps, one branch per route with its condition, and the split's default flow as the `else`. The chain keeps the document order of the routes. Operaton also tests the routes of an exclusive split in document order [Ope26a]. For an exclusive split, the chain therefore routes like the original whenever one of its conditions holds or the split has a default flow. For a parallel, inclusive, or event-based split, the chain cannot express that several routes run, or that the first event decides. The printer therefore writes a *repair comment* above the chain, which asks the developer to repair the split by hand, and reports the split. Listing 4.7 shows this for a parallel fork whose two branches meet in an exclusive merge instead of a parallel join, so that no fork pattern applies.

List. 4.7: The printed source for a parallel fork `Fork` into the tasks `A` and `B`, whose branches meet in an exclusive merge before `C`.

```
process p {
  start S
  // unstructured region: hand-repair required (split Fork degraded to jumps; was parallel)
  if (true) {
    goto A
  } else {
    goto B
  }
  step A
  step C
  end E
  step B
  goto C
}
```

The exclusive merge needs no statement of its own, since `step A` and `goto C` both lead to `C`. Compiled again, the chain sends a token to `A` only, whereas the fork ran both branches, and the repair comment tells the developer that the chain does not express this. A jump refers to a statement, so an edge into a gateway that still chooses between branches, or into an element the printer omits, has no `goto` form. Such an edge is dropped, marked by a repair comment, and reported. The printer therefore always produces source that parses. It is lossy, however: the source may fail validation, and some edges and elements are dropped. Import warns about most places where the recompiled source runs differently. The models that a round trip changes without a warning are reported in Section 6.7.

Limits. The printer gives no modularity guarantee in the sense of Vanhatalo et al. [VVK09, p. 793], for whom a local change of the graph causes only a local change of the decomposition: which rule applies to a split depends on the rest of the graph.

Considered alternatives. A printer built on the RPST (Section 2.3) is also considered (ADR-0014). It would print every fragment with one entry and one exit between two gateways of the

same kind as a block, and so limit `goto` to the remaining fragments, a bound that the catalogue does not give. The catalogue is chosen because it takes less effort to implement and test, and because it rests on dominators, with which Choi et al. [CHKH20, pp. 614, 616] find single-entry single-exit regions that the existing RPST methods miss. The RPST printer is left for future work (Section 7.4.6).

4.5.3 Diagram Interchange Handling

DI data, the coordinates, sizes, and edge waypoints that let a BPMN diagram be redrawn, is dropped in full on import, since the DSL carries no layout (Section 4.1). On compilation, the auto-layout library `bpmn-auto-layout` [bpmn.io26] regenerates the layout entirely. Two ways of keeping a layout are considered (ADR-0009). Layout hints in the syntax would put coordinates into a language meant to describe process semantics. A side channel, either annotations in the source or a separate file that maps identifiers to coordinates, can restore only the shapes of elements with an authored name, since the compiler synthesizes the identifiers of gateways, sequence flows, and boundary events (Section 4.3.4). The price of regenerating the layout is that a round trip loses every layout arranged by hand (Section 7.4.4). Figure 1.2 shows such a generated layout. When the library cannot lay out a process, the document is written without diagram data and with a warning, and Operaton deploys it unchanged, since execution does not require the diagram (Section 2.1).

4.5.4 Equivalence Relation

A round-trip evaluation needs a notion of what a round trip must preserve, defined before the evaluation is run. Since the layout and the synthesized identifiers are regenerated, the relation compares models instead of text.

What is compared. The relation compares the fields of the IR. Table 4.5 lists them per node kind, and any difference in them, or a node gained or lost, is a divergence. The printer has no place for the documentation of a gateway and drops it with a warning, so a documented gateway counts as a divergence. Names and documentation are compared although they do not change how the engine routes a process, so the relation is stricter than execution requires: a round trip that changes only the name of an activity is a divergence.

What is tolerated. The relation tolerates ten differences (Table 4.6). None of them changes which flows a new process instance takes, as the engine code cited for each one in Appendix E shows. A gateway counts as *without settings* if it carries no engine setting, such as a job setting or an execution listener, and no documentation. An identifier counts as *authored* if its element has a name slot and the identifier does not have the form that the compiler generates for an empty name slot (Section 4.3.4).

Table 4.5: Fields compared by the equivalence relation, per node kind of the IR.

Node kind	Compared fields
every node	kind; identifier if authored (tolerance (i)); name and documentation for the process, start and end events, activities, and gateways (not gateway names)
process	identifier, name, documentation, version tag, history time to live, candidate starters, whether the task list offers it for starting, declared error codes with names and messages, declared escalation codes with names
every activity	job settings (asynchronous continuations, exclusivity, job priority, retry cycle), execution listeners, input and output mappings, multi-instance clause with its per-run job settings
start, end, throw, catch	event definition with all its payload (code, message or signal name, timer, condition, link name, variables the payload is bound to) and the documentation of a timer or condition; interrupting flag (start); form fields and initiator (start); implementation binding (end, throw); job settings and execution listeners
boundary event	host (mapped by the bijection), event definition, `cancelActivity`, job settings, execution listeners
user task	assignee, candidates, form key or reference, form fields, due and follow-up date, priority, task listeners
service, send, business rule task	element kind, complete binding (class, expression, delegate expression, external topic, decision, built-in mail or shell behavior), result variable
script, receive task	script format and body as text, result variable; message name
sub-process	kind (embedded, transaction, event sub-process) and its nested flow container
call activity	called element and binding, business key, variable mappings, variable mapping delegate
gateway	kind, job settings, documentation, default flow
sequence flow (edge)	source, target, condition and its documentation, default mark; for an exclusive gateway, the order of its non-default outgoing flows

Content dropped before comparison. The XML reader drops some content while it builds the first model. The relation cannot see this content, so the reader's warnings count it instead. Diagram data is regenerated (Section 4.5.3). Standard attributes without effect in Operaton, such as the `gatewayDirection` of a gateway, and attributes of foreign namespaces on carried elements are dropped without a warning. Content without runtime effect in Operaton, such as lanes and data objects, is dropped with a warning (Section 4.2.1). So are the names and documentation of boundary and intermediate events, since the DSL has no slot for them. Engine settings outside the IR, such as an execution listener on a sequence flow or a form key on the start event of the process, are also dropped with a warning, although Operaton reads them. This last group changes what the engine runs, and it is the largest threat to the claim of behavior preservation (Section 6.7).

The relation. Let *rd* denote reading, *pr* printing, *low* parsing followed by lowering, and *ser* serialization (Figure 4.4). For a document x, the round trip yields $x' = \mathit{ser}(\mathit{low}(\mathit{pr}(\mathit{rd}(x))))$,

Table 4.6: The ten tolerances of the equivalence relation.

	Tolerated difference
(i)	only authored identifiers are compared
(ii)	gateway names are ignored
(iii)	a gateway without settings that passes every token straight on is removed, and its incoming flows lead to its successor instead
(iv)	nodes and flows are compared without regard to their order, except the order of the non-default flows leaving an exclusive gateway
(v)	expressions in conditions, timers, code bindings, field injections, and multi-instance clauses are compared by their syntax tree under the expression grammar of the DSL
(vi)	a missing history time to live counts as the default `P30D`, and the job flags `asyncBefore`, `asyncAfter`, and `exclusive` count as absent when they equal the engine default
(vii)	a none end event with a generated identifier and without a name, documentation, or settings is removed if each flow into it is unconditioned and is the only outgoing flow of an activity or event
(viii)	several flows that enter or leave an activity or event directly count as the gateway that behaves the same way
(ix)	an unconditioned last flow of an exclusive gateway without a default flow counts as its default flow if every earlier outgoing flow is conditioned
(x)	a parallel fork without settings that a single unconditioned flow from another parallel gateway enters, and whose outgoing flows are unconditioned, is merged into that gateway

and it is correct if $rd(x) \approx rd(x')$. For a source text s, the round trip in the opposite direction is correct if $low(s) \approx low(pr(rd(ser(low(s)))))$. The relation $\approx$ compares two models through their reductions. The reduction ρ applies the tolerances listed above in a fixed order (Appendix E): first those that act on the whole model, then those that rewrite each flow container, and last the removal of generated end events (vii). Since the steps run in a fixed order and each step is deterministic, $\rho(M)$ is unique. A reduced model is a labeled directed multigraph per flow container, with the process and every sub-process carrying the fields of their own row of Table 4.5 as a label. The nodes are the flow elements, each labeled with its fields from Table 4.5 except the identifier. The edges are the sequence flows, each labeled with its condition, the documentation of that condition, and whether it is the default flow of its source. Then $M \approx N$ holds if some bijection between the nodes and between the edges of $\rho(M)$ and $\rho(N)$ has five properties: (a) it maps the elements of each flow container into the corresponding flow container; (b) it preserves every label, where a label that refers to another element, such as the host of a boundary event, the default flow, or a nested flow container, is compared after mapping that element through the bijection; (c) it preserves the ends of every edge; (d) it keeps the order of the non-default flows leaving each exclusive gateway (iv); and (e) a node carries an authored identifier exactly when its image carries the same one (i). The identity is such a bijection, the inverse of one is one, and so is the composition of two, so $\approx$ is an equivalence relation, and since both models are finite, it is decidable. Section 6.3.3 describes how the evaluation decides the relation.

In terms of lenses [FGM+07, pp. 6, 9, Defs. 3.1, 3.2, 3.7], compilation is the get direction from source text to document, and import is the putback of an oblivious lens (Section 3.10). The round trip of a source text is then the first round-trip law of Section 3.10, GetPut, and the round trip of a document the second, PutGet [FGM+07, p. 6, Def. 3.2]. Here both laws can hold only up to $\approx$: an oblivious lens that satisfies them exactly has a bijective get direction [FGM+07, p. 9,

Lemma 3.9], and compilation is no bijection, since it regenerates the diagram and the synthesized identifiers, and printing loses formatting and declared types. This thesis tests the two laws on samples and does not claim that the pair forms a quotient lens, a lens whose laws hold modulo an equivalence [FPP08, Sect. 2]. The relation is syntactic: it compares conditions by their syntax tree or text, so two models that route alike through differently written conditions are not equivalent. We do not prove that $\approx$ implies equal behavior.

The compiler, the importer, and the language server that realize this design are described in Chapter 5.

5 Implementation

This chapter describes how the design of Chapter 4 is realized in code. Section 5.1 gives the overall structure. The following sections trace a process through the tool. They cover the front end that parses and checks the source (Section 5.2), the editor integration (Section 5.3), the compiler back end (Section 5.4), and the import (Section 5.5). The chapter closes with the command-line tool, the use of generative AI, and the tests (Sections 5.6 to 5.8). Chapter 6 evaluates this implementation.

This chapter describes release 0.1.1 of BPMNscript [Kra26b], which is published under the Apache License 2.0. The implementation is written in TypeScript and builds on the language workbench Langium 4.4.0 [Langium26b] and on two libraries of the bpmn.io project, `bpmn-moddle` 10.3.1 [Reh26a] and `bpmn-auto-layout` 1.3.0 [bpmn.io26].

5.1 Architecture Overview

The implementation is one repository with five packages. Four of them make up the tool and are divided at the IR of Section 4.4, and the fifth holds the integration and engine tests. Table 5.1 lists them with their role and size. How they depend on each other and on the libraries they build on is shown in Figure 5.1.

Table 5.1: The five packages of the implementation, with the lines of hand-written TypeScript source in each (generated code and tests excluded). The `tests` package holds only tests and is therefore not counted.

Package	Role	Lines
`language`	grammar, parser configuration, validator, and the services of the language server	9,780
`transform`	the IR and the four transforms between source, IR, and XML	18,637
`cli`	the command-line tool `bpmns`	368
`extension`	the Visual Studio Code extension: language client and conversion commands	660
`tests`	round-trip, fuzz, and engine tests across packages	–

The `language` package knows the syntax and nothing about XML: it holds the grammar, the validator, and the services that the editor asks for completion or hover information. The `transform` package knows the IR and XML and holds the four transforms of Figure 4.4. Lowering, the transform from source to IR, reads the syntax tree (the AST, Section 2.6) that the `language`

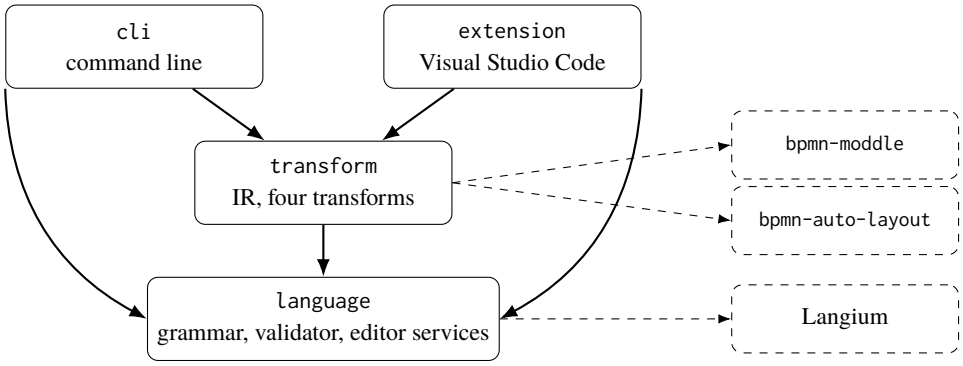


Figure 5.1: The four packages and the libraries they build on. A solid arrow points from a package to a package it imports, a dashed arrow to the main library on which a package builds. The `tests` package, which imports `language` and `transform`, is left out.

package produces. The other transforms take the vocabulary of the language from the same package. `transform` therefore imports `language`, and never the other way round. The command-line tool and the extension are the *hosts* of the transforms: they only read files, call the transforms, and report diagnostics and warnings.

One module of the `language` package, the *vocabulary module*, holds the soft keywords of Section 4.3.3, such as trigger kinds, setting keys, and listener events, together with the settings that each kind of statement takes. The validator, the completion provider, and the XML reader and printer of the import (Section 5.5) take these lists from it instead of spelling the words a second time.

5.2 Language Front End with Langium

Five candidates are assessed in depth (ADR-0003): the language workbenches Langium, Xtext, MontiCore, and JetBrains MPS, and the parser generator ANTLR with a hand-written language server. Langium is chosen mainly because it runs on the same platform as the two libraries the transforms build on, and its tool chain is also judged easier to learn than that of Xtext. Langium, `bpmn-moddle`, and `bpmn-auto-layout` all run on Node.js, so the compiler calls the two libraries directly, whereas Xtext and MontiCore run on the Java Virtual Machine (JVM) [CHR26, p. 127] and would need a bridge to these libraries or replacements for them. The other two candidates fall short on editor support (Section 4.1). On top of an ANTLR parser, every editor feature would be written by hand, whereas Langium is built around the LSP (Section 2.6). JetBrains MPS is excluded because it is a projectional editor that saves the syntax tree itself, mostly in XML files [Voe13, p. 68], and its editor is a proprietary solution [CHR26, p. 126]. The main drawback of Langium for this work is that it serializes a syntax tree only to JavaScript Object Notation (JSON) [Langium26b] and has no serializer back into source text. The printer (Section 4.4) is therefore written by hand (Section 5.5.2).

5.2.1 From Grammar File to Parser

A Langium language starts from a grammar file with parser rules, which describe the nodes of the syntax tree and their properties, and terminal rules, regular expressions that cut the text into tokens. The Langium generator turns the parser rules into TypeScript interfaces, one per node type, so that every later component works on typed nodes [Langium26a; Langium26e].

After parsing, Langium indexes the names a document declares, computes scopes, links and indexes cross-references, and validates the document [Langium26d]. Each of these phases is the job of a *service*, an exchangeable component of the language implementation, and a language replaces a default service by registering its own under the same name [Langium26c]. The implementation replaces the services for scoping, linking, references, validation, the error messages of lexer and parser, and the editor features, and keeps the Langium defaults for others, such as folding.

Listing 5.1 shows three rules that illustrate how the grammar of Section 4.3.3 is written as a Langium file. In Langium's notation, `name=Rule` assigns what a rule parses to a property of the node, `+=` appends it to a list, `[Statement:Name]` is a cross-reference to a node of type `Statement` written as a name, `{infer Setting}` creates a node of type `Setting`, and `?` and `*` mark optional and repeated parts.

List. 5.1: Three rules of the grammar file, with comments omitted and one long line broken.

```
fragment SettingsParens:
    '(' items+=ParenItem (',' items+=ParenItem)* ')';

ParenItem:
      {infer Setting} key=ParenKey ':' value=Expr
    | {infer Flag} flag=FlagWord
    | {infer ParenValue} value=Expr;

OnHandler:
    'on' (host=[Statement:Name] ':')? trigger=ID
    SettingsParens? AttributeBlock? body=Block;
```

Keywords. A quoted word in a Langium grammar, such as `'on'`, becomes a keyword everywhere in the file and can then be used as a name only where the grammar admits it explicitly [Langium26f]. The grammar therefore reserves only the 42 keywords of Section 4.3.3. It reads every soft keyword as a plain identifier (`trigger=ID` in Listing 5.1).

Settings. The rule `ParenItem` in Listing 5.1 accepts all three kinds of items of Section 4.3.3 (keyed settings, flags, and the payload) in one list. The grammar therefore leaves open which settings a user task or a timer takes. That knowledge sits in a table of the vocabulary module with one row per kind of statement, which the validator and the completion provider both read. This keeps the grammar small, and an unknown key produces a message that identifies the key

and the statement, such as `setting 'asyncBefor' is not valid on a service task`, instead of a syntax error.

5.2.2 Scoping and Cross-Reference Resolution

A cross-reference is a grammar slot written as a name and resolved after parsing to the node of that name, such as `host=[Statement:Name]` in Listing 5.1. The grammar has four of them: the target of a `goto`, the host of a handler, the code of an error mapping, which maps a failure of an external task to an error code, and every identifier inside an expression. Two Langium services resolve them [Langium26d]: for each reference, the *scope provider* decides which nodes it may resolve to, and the *linker* looks the name up among them.

By default, Langium resolves a name against the blocks that enclose the reference, from the innermost outward [Langium26d]. That rule fits neither `goto` nor `on Host:`. It misses legal targets: a statement inside an `if` block is a legal target for a `goto` after the block, but the default rule does not see it. It also finds illegal ones: from inside a sub-process, the default rule sees the statements around it, although no sequence flow may cross the sub-process border (Section 4.4). The custom scope provider therefore implements the flow-container rule of Section 4.3.4: it collects every statement with a name in the flow container that holds the reference, at any depth of nesting, and nothing beyond.

When a lookup fails, Langium's default message states the grammar type it looked for. The custom linker instead searches the whole process for a statement of that name and reports either that none exists or where it is, as in `'Deep' is inside subprocess 'Inner'`.

Expressions need a second mechanism. The bare word in `throw error(OUT_OF_STOCK)` refers to an error code, and the bare word in `if (amount > 500)` refers to a variable. Both are the same token in the same position, so the parser cannot tell them apart. The grammar therefore reads both as a reference to a code, and the scope provider resolves it only where Section 4.3.4 allows a code. The custom document validator drops the failed links of variables, so an undeclared variable produces only the warning of Section 4.3.4.

Variables thus never link, but the editor still offers navigation for them (Section 5.3.2). Langium's go-to-definition, find references, rename, highlighting of occurrences, and hover all start by asking the references service for the declaration under the caret and for its uses. The custom references service answers for variables itself: it returns as declarations every place that declares the name (Section 4.3.4) and as uses every identifier in the process that is read as that variable.

5.2.3 Validation and Diagnostics

Four sources of diagnostics. A diagnostic, an error or warning attached to a range of the source, comes from the lexer, the parser, the linker, or the validator, which Langium runs in that order [Langium26d]. The lexer reports characters that form no token, the parser reports text that does not fit the grammar, the linker reports a statement or code name that does not resolve, and

the validator reports everything that parses and links but is still wrong. Every source runs even after an earlier one has reported an error: after a syntax error, the parser recovers and hands on a partial syntax tree, and the validator still checks it, so a syntax error can come with follow-up errors. Listing 5.2 shows three of the sources on one small file.

List. 5.2: A faulty source and the diagnostics `bpmns build` prints for it, abridged, in the order Langium produces them: parser, linker, validator. The messages use "step" for any statement.

```
process p {
  start S
  service A(label: "no binding")
  goto Nowhere
  user B(assignee: "x"
}
```

```
line 6: expected ')' before '}' [}]
  note: a '(' is still open
line 4: no step named 'Nowhere' in this process [Nowhere]
line 5: this step can never run [user B(assignee: "x"]
  note: an earlier 'end', 'throw', 'goto', ...
line 3: service task 'A' has no 'class', 'expression',
  'delegate', 'topic', or 'type' setting [A]
```

Engine rules. Some of the checks repeat a rule by which Operaton's deployment parser refuses a document (Section 4.1), and their messages state in a note that the engine refuses the construct. For example, a process may have only one plain or timer start, and a second one produces the message of Listing 5.3. It corresponds to the parser's own error "multiple none start events or timer start events not supported on process definition" [Ope26a].

List. 5.3: The validator's message for a second plain or timer start in a process.

```
this is the process's second plain or timer start
  note: Operaton refuses a second one, so the deployment fails
  help: keep one, or give this start a message, signal, or condition trigger
```

These checks cover the rules identified during the design, and we do not prove that they cover every refusal of the engine. Leaving these rules to the engine would avoid restating rules that can change with the engine, but the author would then meet a broken rule as a failed deployment, far from the line that breaks it (ADR-0013). For the mirrored rules, a refusal of the engine is an error, and a setting the engine ignores is a warning, since the process still deploys and runs. An example of the second kind is a retry cycle on an activity without an asynchronous continuation, since the engine then creates no job for the cycle to configure (Section 4.3.5) [Ope26a]. The validator also has rules of its own that are stricter than the engine. Operaton accepts a service task with two bindings and uses the one that comes first in a fixed order of `type`, `class`, `delegateExpression`,

and `expression` [Ope26a], whereas the validator refuses it, because the engine then ignores one of the two bindings the author wrote. In the same way, an unreachable statement is an error, because it would compile to a node without an incoming flow, and the engine never runs such a node.

Messages. Where it can, a message says what the author probably meant and what to write instead, as the examples of Listing 5.2 show. A misspelled trigger, listener event, parameter direction, or type is answered with the list of legal words at that position, and a misspelled setting key is reported as not valid on that kind of statement.

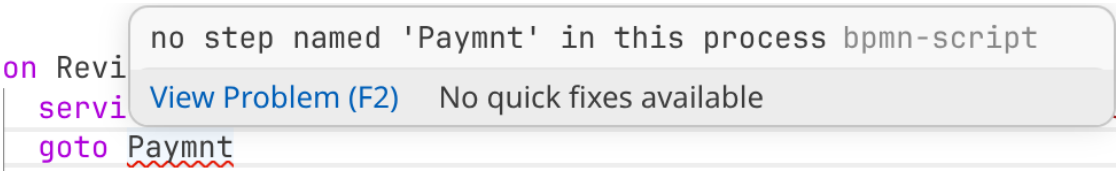
Hosts. The command-line tool, the language server, and the conversion commands of the extension all create the same set of services and build documents with the same Langium document builder, so they produce identical diagnostics. They differ only in how they show them: the editor underlines every range at once, whereas `bpmns build` prints only the errors when there are any, and the warnings otherwise (Section 5.6).

5.3 IDE Integration via LSP

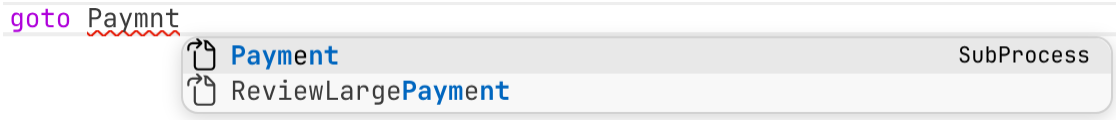
The editor support consists of two processes. The extension runs in the extension host of Visual Studio Code, the process in which the editor runs extension code. When it is activated, it starts the language server as a separate Node.js process and talks to it over inter-process communication with the LSP. The language server only creates the services of the `language` package and starts Langium's server loop, so every editor feature is a service of the `language` package or a Langium default. Table 5.2 lists the editor features and where each comes from. Visual Studio Code is the primary target (ADR-0004). The server uses only the LSP, so another editor with an LSP client can start it and receives every feature that Table 5.2 marks as LSP. The TextMate highlighting and the conversion commands are specific to Visual Studio Code, and no other editor integration has been built or tested. Figure 5.2 shows diagnostics, completion, and hover on the full order-handling example of the repository [Kra26b], excerpted in Listing 4.1.

5.3.1 Syntax Highlighting and Semantic Tokens

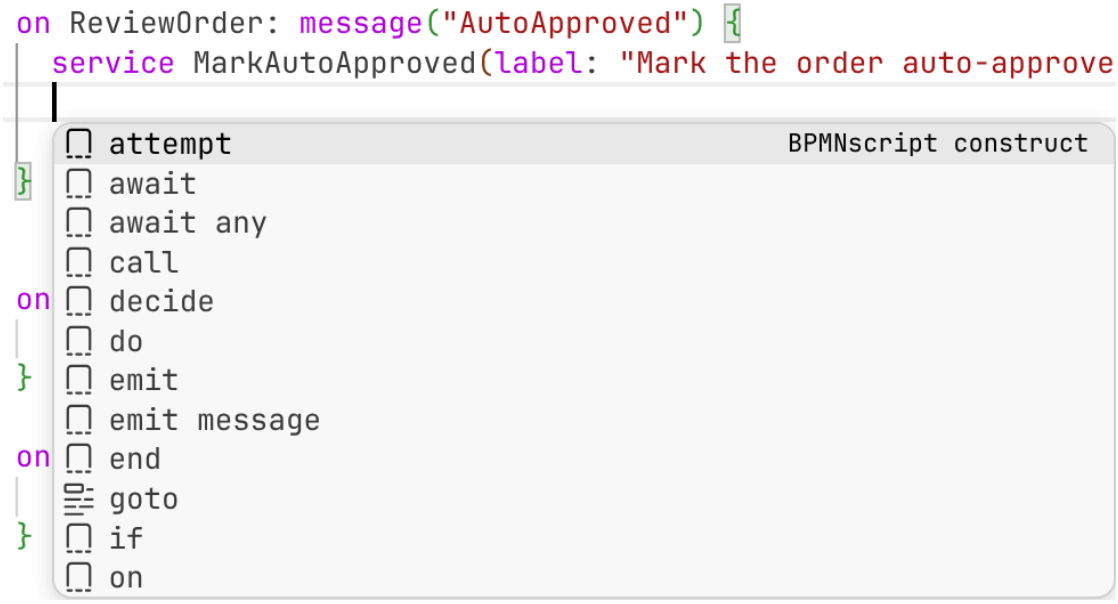
Highlighting comes in three layers. The first is a TextMate grammar, a JSON file of regular expressions with which the editor colors text without parsing it. Langium generates it from the grammar [Langium26e]. The second layer is semantic tokens, colors that the language server computes from the syntax tree. They are needed because the soft keywords of Section 4.3.3 are plain identifiers to the TextMate grammar, which cannot tell the trigger in `on error(...)` from the variable in `var error: string`. The semantic-token provider gives soft keywords the keyword token type where the syntax tree shows them in a keyword role. The third layer is a hand-written injection grammar, which colors the body of a fenced script in the language given after the opening



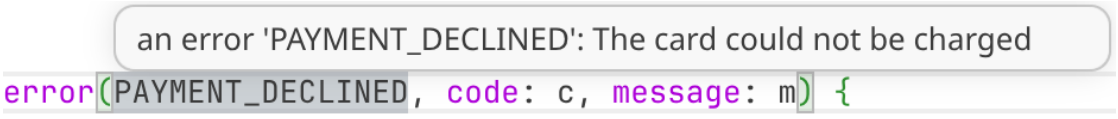
(a) Diagnostic on a misspelled jump target



(b) Completion of the jump target



(c) Completion at the start of a statement



(d) Hover on an error code

Figure 5.2: Editor support in Visual Studio Code on the full order-handling example. (a) The target of `goto Payment` is misspelled as `Paymnt`, and the editor underlines it and reports `no step named 'Paymnt' in this process`. (b) Completion at the misspelled target offers the statements of the process whose names match it. (c) On an empty line inside a handler, completion offers the statement keywords. (d) Hovering over the code `PAYMENT_DECLINED` shows the kind of the code and the message of its declaration.

Table 5.2: The editor features, the channel they use, and whether the implementation provides them or uses the Langium default.

Feature	Channel	Implementation
diagnostics	LSP	lexer, parser, linker, and validator (Section 5.2.3)
completion	LSP	custom provider over the tables of the vocabulary module
hover	LSP	custom provider plus documentation comments
go-to-definition, find references, highlighting of occurrences	LSP	Langium defaults over the custom references service
rename	LSP	custom subclass of the Langium default that writes a new name in backticks where it needs them, over the custom references service
outline	LSP	custom, with handlers as entries
folding	LSP	Langium default
semantic tokens	LSP	custom, for words whose role depends on position
base highlighting	TextMate grammar	generated by Langium, then adjusted
script bodies	TextMate injection	hand-written, for Groovy, JavaScript, Python, and Ruby
conversion commands, sidebar	extension API	`transform` package in the extension host (Section 5.6)

fence, such as `groovy`. The fence is only highlighted, and no completion or diagnostics reach inside it, since the parser reads the whole fence as one token.

5.3.2 Completion, Navigation, Hover, and Outline

Completion. Langium's default completion offers the keywords and cross-references that the grammar allows at the caret. For the DSL this is not enough, because most of the vocabulary consists of soft keywords, which the grammar sees as plain identifiers, and it is too much, because the grammar allows a keyword in places where the validator refuses it. The custom provider therefore offers what the validator accepts at the caret and reads its word lists from the same tables. At the start of a statement, it offers a snippet for each statement kind, a template whose fields the author fills in one after another, such as `user id(assignee: "user")`. Inside the parentheses, it offers the setting keys of that kind of statement and, while a required payload is missing, only candidates for the payload, such as the declared error codes after `error(`. After `on`, it offers as hosts only activities and as triggers only those that the host permits. In a condition, it offers the process variables with their types. Every snippet the provider offers is tested: inserting it must produce source that parses and validates without errors.

Navigation. Go-to-definition, find references, rename, and highlighting of occurrences work for the statements that `goto` and `on Host:` refer to, for declared error and escalation codes, and for process variables. The tests require that the source still parses and validates after each rename.

Hover and outline. Hovering over a statement, a code, or a variable shows its kind, name, and label or type, such as `a user task 'Review': Review the invoice`. A documentation comment,

`/** … */`, above the declaration follows as a paragraph of its own. The hover is the same on a declaration and on every reference to it, because Langium resolves the reference first. The default outline skips handlers, which have no name, so a custom provider lists them.

5.4 Compiler Back End: BPMN 2.0 XML Generation

The back end runs two transforms. Lowering turns the checked syntax tree into the IR, and serialization writes the IR as a BPMN 2.0 XML document with a generated diagram. The hosts call them only after validation has passed.

5.4.1 Lowering the Syntax Tree to the IR

The IR of Section 4.4 is a set of plain TypeScript types. A sequence flow refers to its source and target by identifier, and a boundary event refers to its host in the same way. The IR stores the job flags `asyncBefore`, `asyncAfter`, and `exclusive` only when they differ from the engine's default, so a document that states the default and one that omits it lower to the same IR.

Lowering handles the statements of a body in one pass from top to bottom. Each statement is lowered on its own and reports two node identifiers. Its *entry* is the node into which the flow from the previous statement leads, and its *exit* is the node from which the next statement continues. The pass then adds one sequence flow from the exit of each statement to the entry of the next. This is how two consecutive statements become connected without a written sequence flow (Section 4.3.3). For a task, entry and exit are the task itself. For an `if` or a `parallel` block, the entry is the split gateway and the exit the join, with the branches lowered in between by the same procedure, and a `while` loop enters and leaves through its loop gateway. A loop also reports the identifier of its gateway's default flow, so that the flow from the loop to the next statement is that default flow. An `end`, a `throw`, or a `goto` reports no exit, so no flow leads from it to the next statement. A `goto` creates no node at all and reports the jump target as its entry, so that the predecessor's flow leads straight there. The default flows and joins of the branches follow the rules of Section 4.5.1.

The synthesized identifiers follow fixed templates (Section 4.5.1), and the printer and the round-trip comparison use them to recognize generated identifiers.

A condition reaches the IR as the text the engine evaluates. A function of the `language` package prints the expression tree as `${amount > 500}`, with parentheses only where the author wrote them, and passes a raw template through as written. Settings that the engine reads as plain text, such as the name of a delegate class, stay as written.

5.4.2 IR-to-XML Mapping Rules

Serialization builds the document as an object tree with `bpmn-moddle` instead of writing XML text. `bpmn-moddle` knows the BPMN metamodel from a JSON description of its element types and attributes [Reh26a]. It creates a typed object for each element, resolves references between objects, and serializes the tree as namespaced XML. Reading works the other way round. An extension namespace, such as `operaton:`, is added by a further description of the same kind (Section 5.4.3).

Serialization first creates the root elements, the elements outside the process that events refer to. There is one `error` or `escalation` per code and one `message` or `signal` per name (Section 4.5.1). Every event can then refer to its root element as soon as the event is created. It then creates the process element and, for each flow container, every flow element and every sequence flow, and sets the references between them, such as the incoming and outgoing flows of every element, the default flows, and the hosts of boundary events. Before the layout (Section 5.4.4), serialization reads its own output back with `bpmn-moddle` and fails on any warning, because the layout library would otherwise drop an element with a duplicate or illegal identifier without an error [bpmn.io26; Reh26a].

Modeling tools show the name of an element. An activity without a `label` and the process itself therefore receive a name derived from their identifier, so that `ReviewInvoice` is shown as “Review Invoice.”

Of the five job settings of Section 4.3.5, four are written as attributes, `operaton:asyncBefore`, `operaton:asyncAfter`, `operaton:exclusive`, and `operaton:jobPriority`, and the retry cycle as the extension element `operaton:failedJobRetryTimeCycle`.

5.4.3 Operaton Extension Attributes

The description of the `operaton:` namespace is derived from `camunda-bpmn-moddle` [Reh26b], the description of the Camunda namespace for `bpmn-moddle`, with the namespace changed and the content cut down and adjusted (ADR-0008). It lacks mainly attributes without a setting in the DSL, such as tenant identifiers, case references of call activities, and form keys on start events. Connectors, through which a task calls an external service such as a REST endpoint (Section 6.6), and other types that only the XML reader needs remain, because the XML reader has to recognize what a document carries to refuse or report it. The Camunda description is not loaded beside the Operaton one, since both define attributes such as `class` and `assignee`, and `bpmn-moddle` drops every element whose type receives the same property from two descriptions [Reh26a]. Documents in the Camunda namespace are handled on import instead (Section 5.5.1).

5.4.4 Auto-Layout for Diagram Interchange

`bpmn-auto-layout` takes a document as text and returns it with a new diagram: it reads the document, discards any diagram data it contains, places every element on a grid, and draws the edges [bpmn.io26]. The only information it reads from an existing diagram before discarding it is whether a sub-process is drawn expanded, and it lays out every other sub-process collapsed. Before the layout, serialization therefore adds a hint to the document, a minimal diagram that contains one shape per sub-process, marked as expanded and without coordinates.

The library can fail on a process and throw an exception. Serialization then removes the hint again, because the engine refuses a diagram without coordinates [Ope26a], and the hosts write the document without a diagram (Section 4.5.3). The generated layout has defects that affect only the drawing in a modeling tool, since Operaton does not execute the diagram data. For example, an edge from a `goto` into a sub-process can cross the children of the sub-process, two boundary events on a narrow host can overlap, and the exit flow of a loop can be drawn on the same line as the flow back into the loop (Figure 1.2).

5.5 Import: BPMN 2.0 XML to DSL

Import runs in the two steps of Section 4.4. The XML reader reads a document into the IR and either refuses it or returns the IR together with a list of warnings, and the printer prints the IR as source, again with warnings.

5.5.1 Reading: Refusals, Warnings, and the Camunda Namespace

Steps. Before it parses a document, the XML reader checks the raw text for what `bpmn-moddle` would hide or coerce, such as a document type declaration, which the engine refuses [Ope26a]. It also handles the Camunda namespace on the raw text, as the end of this subsection explains. It then reads the document with `bpmn-moddle`, using the same Operaton description as the compiler, and turns the library's warning about a duplicate or illegal identifier [Reh26a] into a refusal. It walks the process recursively and maps each element to its IR node, refusing an element type for which it has no mapping. After the elements of a flow container, it checks the rules that involve several elements, such as the outgoing flows of an exclusive gateway, the hosts of boundary events, and the pairs of link events.

Refusals and warnings. A refusal message identifies the element and what makes it impossible to import. Where the engine itself refuses the construct, a note identifies the check in the engine's deployment parser that refuses it, in some cases with the engine's own sentence. An exclusive split with two unconditioned flows and no default flow, for example, produces the refusal of Listing 5.4, where `validateExclusiveGateway` is the deployment parser's check of exclusive gateways [Ope26a].

List. 5.4: The import's refusal of an exclusive split with two unconditioned flows and no default flow, abridged and with one long line broken.

```
exclusive gateway 'G' cannot be imported
  note: Operaton's 'BpmnParse.validateExclusiveGateway' fails the
  deployment with "Exclusive Gateway 'G' has outgoing sequence flow
  'fa' without condition which is not the default flow." ...
```

A warning reports content that the import drops or rewrites. It also reports content that the import carries over although the rebuilt process may run differently. Warnings fall into categories such as a dropped lane, dropped documentation, a rewritten construct, and changed behavior.

To report the engine settings it drops (Section 4.2.1), the XML reader keeps a table per element type of the Operaton attributes and extension elements that the mapping consumes. After mapping an element, a sweep compares every piece of Operaton content on it with this table and warns about every piece left over, including Operaton attributes unknown to the XML reader. A further table lists, for settings that Operaton reads but the IR has no field for, the engine method that reads them, so that the warning can say that the rebuilt process runs without the setting. Attributes of the BPMN schema are not swept, so the XML reader checks a fixed list of them, such as `startQuantity`, and drops the others without a warning (Section 4.5.4).

The Camunda namespace. Documents written for Camunda 7 use the namespace `camunda:`, which Operaton reads as a fallback for `operaton:` (Section 2.5). The XML reader reaches the same result in two steps on the text, before `bpmn-moddle` parses it. First, where an element carries a setting in both namespaces, the XML reader removes the Camunda spelling, an attribute or an extension element, and reports it in a warning, because the engine reads only the Operaton one. For `operaton:class="A"` beside `camunda:class="B"`, the import therefore keeps `A`, which is the class that the engine runs. Second, the XML reader replaces the Camunda namespace Uniform Resource Identifier (URI) by the Operaton one where the document declares it. `bpmn-moddle` binds an element to a type by its namespace URI [Reh26a], so every Camunda element and attribute then arrives as an Operaton type. The document produces one warning for the rewrite, and the rebuilt document uses `operaton:` alone.

5.5.2 Restructuring the Graph into Blocks

The printer turns the flat graph of each flow container into nested statements by the pattern catalogue of Section 4.5.2. It consists of two parts: an analysis of the flow graph and the printing walk.

The flow graph. For each flow container, the analysis builds a graph whose nodes are the flow elements and whose edges are the sequence flows. A virtual entry node is added with an edge to every start event, every boundary event, and every link catch event without an incoming flow,

and a virtual exit node with an edge from every end event and every node without an outgoing flow. A sub-process is a single node of its parent's graph and receives a graph of its own. On this graph, the analysis computes dominators and post-dominators (Section 2.3). Both relations are computed by the iterative procedure of Cooper et al. [CHK01], the post-dominators as the dominators of the reversed graph, after every loop without a path to the exit has been given an edge to it. The procedure also works on irreducible graphs [CHK01, pp. 5–6, Fig. 2], in which a loop can be entered from outside at more than one node [HK04, p. 319]. A `goto` into a loop body produces such a graph.

Loops. Finding loops as edges back to a node that dominates their source presumes loops with a single entry. The analysis therefore finds loops as strongly connected components, maximal sets of nodes that can all reach one another. For each component, it chooses as head a node that is entered from outside, preferring the gateway of a `while` test and then the body entry that a `do` test returns to. Every edge from inside the component into the head becomes a back edge, and the search repeats on the component without its head to find nested loops. These back edges are what the two loop patterns of the printer (Appendix F) fold into `while` and `do … while`.

The printing walk. For each flow container, the printer walks the graph from each entry, prints each node it reaches, and continues with its successor. Each kind of node is printed in its own way: an exclusive gateway is first tried as a loop and otherwise printed as an `if` chain, a parallel or inclusive gateway as a `parallel` block, and an event-based gateway as an `await` block. The printer decides where the branches of a split end by six continuation rules for an exclusive split, and by the merge the split reaches for a parallel, inclusive, or event-based split (Appendix F). The printer records which nodes and flows it has printed. A walk that reaches a printed node prints a `goto` to it instead of printing it again, and after all walks, every unused flow is printed as a jump as well. A `goto` must refer to a statement, and a gateway is not one. A jump into a gateway with only one way left out of it is therefore forwarded through the gateway to the statement behind it. A jump without such a target, into a gateway that still chooses or into an element the printer omits, is dropped, and the printer writes a repair comment in its place (Section 4.5.2).

Declarations and text. To declare every variable that the process reads (Section 4.2.4), the printer parses every condition, every loop setting, and every variable a call activity passes in, and subtracts the names declared in the places of Section 4.3.4. For the running example, the subtraction leaves no variable, because the forms of the start event and of `ReviseClaim` declare both `amount` and `compliant`. Compiling Listing 1.1 and printing the result therefore yields the same statements and no `var` lines. A condition inside the expression sub-language (Section 4.3.3) is printed as an expression, `if (amount > 500)`, and any other condition as a quoted raw template. The XML reader drops the name of an activity when it equals the name the compiler derives from the identifier, so that the printed source states a label only where it differs from the derived name.

Warnings. The printer throws an exception only on a malformed IR, such as two elements with the same identifier. Where the source cannot express the model, the printer reports a print warning, such as a dropped edge, a split degraded to jumps, a dropped condition, or an unreachable activity that it leaves out. The printer does not check whether the printed source passes the validator (Section 5.6).

5.6 Command-Line Tool and Conversion Commands

The command-line tool `bpmns` has two commands. `bpmns build` compiles a `.bpmnscript` file to BPMN 2.0 XML: it builds the Langium document with validation, stops on any error, lowers and serializes it, and writes the result next to the source unless another path is given. `bpmns parse` imports a BPMN file: it reads and prints it, writes the source, reports the warnings of both steps, and then builds the written source once more, reporting its errors with a note that hand-repair is needed.

Inside the editor, the extension offers the same two conversions as the commands “Compile to BPMN” and “Decompile to BPMNscript,” and the second runs the importer. A sidebar view adds one button that converts the open file in the direction its file type calls for and one that picks a BPMN file to import. The conversion runs in the extension host, which creates its own services of the `language` package and calls the `transform` package directly.

5.7 Use of AI-Assisted Coding

Generative AI tools, above all Claude Code with the models listed in Appendix I, took part in the code of BPMNscript and of the evaluation repository. We set up the repository and Langium and wrote parts of the early code ourselves. The model wrote much of the Langium internals and the infrastructure around them, and the majority of the tests. It also wrote much of the feature code, especially later in the project, once the architecture and the patterns were clear. The requirements, the scope of the supported subset, the design of the language and its architecture, and the design of the evaluation are our own. The model’s part was proposing alternatives, challenging decisions and assumptions, working out details, and writing the code.

Tools. In Visual Studio Code, GitHub Copilot provided inline completion, answered questions in its chat, and reviewed the pull requests on GitHub. Planning, implementation, and review ran in Claude Code (models in Appendix I). The agents followed devline [Kra26c], a Claude Code plugin we wrote with Claude Code, which runs brainstorming, planning, test-driven implementation, and review as separate agents. A project instruction file states the conventions every agent follows, such as one parameterized test per behavior and assertions on complete result sets, and was refined throughout the project.

Design. We used the model as a sparring partner for the syntax and the architecture. It checked our assumptions, pointed out gaps, and proposed alternatives from the literature and from other tools, and we decided between them. The model drafted the architectural decision records from these discussions, and we reviewed and corrected them (Appendix H).

Source analysis. The model searched the source code of Operaton [Ope26a] and of `bpmn-moddle` [Reh26a] for behavior their documentation leaves open. Many edge cases of the validator and the tests come from this analysis, such as the 17 combinations of event type and position that the specification does not permit and Operaton deploys with their type ignored (Section 4.2.1). The engine tests check part of this behavior against Operaton 2.1.0 itself (Section 5.8).

Evaluation. We defined what the evaluation measures and how: the research questions, the corpus and its selection criteria, the equivalence relation, and the architecture of the evaluation tools (Section 6.1). The model helped to work out the remaining details and implemented the tools. We reviewed and corrected the tools and then ran the evaluation and checked a sample of its results by hand against the original models.

Guiding the model. Working with the model required guidance, which the project instruction file records. Left to itself, the model wrote one or more tests per method, missed tests of the behavior that mattered, added new code where refactoring the existing code would have served, and produced changes of thousands of lines, too large to review with care. We therefore reduced the tests to one parameterized test per behavior a feature implements and per aspect of each supported BPMN construct. One cleanup pass removed about 4,300 test lines, which made the test suite easier to survey and the diffs of later changes smaller. For the behavior of Operaton, `bpmn-moddle`, and Langium, we wrote detailed documents on which sources answer which question, since the model otherwise assumed behavior it should have checked, such as the order in which Operaton picks one of two bindings on a service task.

Review. Before any change was merged into the main branch, we read it, ran the full test suite and the type check ourselves, since agents at times reported failing checks as passing, and checked that the tests cover the intended behavior. For a large change, this reading covered the behavior of the feature and the critical parts of the code. We also read every change to an existing test with care, so that an altered test could not hide a regression. We tested every new feature as a whole, as well as the core elements the rest of the code depends on. The engine tests and the round trips of Chapter 6 judge the output with Operaton itself, an oracle independent of the model. Since AI was involved in most commits, the commits carry no individual attribution, and this section documents its use for both repositories as a whole.

5.8 Testing Infrastructure

The test suite has 2,552 tests. Of these, 1,833 are package tests of parsing and validation (Section 5.2), the editor services (Section 5.3), the command-line tool (Section 5.6), and each transform on its own (Sections 5.4 and 5.5), 514 are integration tests through the whole pipeline in the `tests` workspace, and 205 are engine tests. This section describes only the mechanisms behind the generated and mined round trips, and their results are reported in Chapter 6.

Golden pairs. A golden file is a file frozen in the repository: a reference output that a test compares against, or a fixed input. The repository holds 22 golden pairs, each a source and the XML that the compiler produces from it, and four golden XML files without a source. Each pair but one covers one feature area of the language, such as boundary events, forms, listeners, repetition, or link events, and the remaining pair covers a bug fix. Each pair holds output that we tested by hand before freezing it. Any later change to the output of a finished area therefore makes a test fail and is accepted only after a review of the diff. For each pair, a test regenerates the XML, compares it byte for byte with the frozen file, and imports it. It then requires the IR compiled from the printed source to equal the IR of the original source, apart from differences that do not change what the engine runs, such as generated identifiers, gateway names, and the order of elements where the engine ignores it.

Engine tests. The engine tests run Operaton 2.1.0 itself. The repository contains a small example application that embeds the engine and can also run the 32 example processes, and the tests start this application in a container. Most engine test files deploy example processes over the engine's REST interface, start instances, and check what happens, for example that an interrupting boundary event cancels its host or that an inclusive join waits only for the branches that were taken. The remaining test file deploys 62 XML files as written and a second time after import and compilation. Two of them are written to be refused: Operaton refuses one as written, and import refuses both. They are the golden files, the XML inputs of other tests, and the compiled examples.

Fuzz test. The fuzz test generates random programs from a seeded generator, over the seeds 1 to 200 in the suite, and sends each through the whole round trip twice. It fails on a crash, a stage that exceeds its time limit, a refused import, a printed source with errors, a changed model, a print warning, a second print that differs from the first, or a reading warning, except the warning about an `attempt` block that ends in `cancel` without a `cancel` handler, which the generator writes. Minimized failing programs are kept as 158 regression tests. Chapter 6 evaluates this implementation on generated programs and on existing BPMN models.

6 Evaluation

This chapter validates the treatment of Chapters 4 and 5 against RQ4 and the conciseness part of RQ3 (Section 1.3). We first state the methods and what counts as a pass for each (Section 6.1). We then report the round trips of generated programs, of real-world models, and of the Model Interchange Working Group (MIWG) reference models (Sections 6.2 to 6.4) and measure the size of the notation (Section 6.5). It then interprets the results (Section 6.6) and closes with the threats to validity (Section 6.7). Chapter 7 answers the research questions from these results.

6.1 Evaluation Design and Equivalence Criteria

In the evaluation, a fixed tool runs over fixed inputs, and every outcome is a count that a rerun reproduces, apart from three originals whose timer start event makes the outcome depend on the date of the run (Section 6.7). It is the treatment validation of the design cycle [Wie14, p. 27], which checks the artifact against the executable models that exist for Operaton before any practitioner uses it. It runs in an evaluation repository separate from the tool [Kra26a], which pins BPMNscript to the release tag `v0.1.1` [Kra26b], builds that release from source, and records the commit of the tool, of the evaluation repository, and the image digest of the engine in every result. The tag `v1.1.0` of the evaluation repository holds the code and the results of this chapter.

Four inputs are measured, each in a stage of its own: (1) programs from the generator of the fuzz test (Section 5.8), which test the round trip from source text; (2) the official example and test models of Operaton and Camunda together with the examples of BPMNscript and the golden files of its test suite (Section 5.8), chosen on purpose; (3) executable models mined from GitHub; and (4) the reference models of the MIWG [OMG26]. The last three test the round trip from a document.

Each document goes through the same steps (Figure 6.1). First, Operaton 2.1.0 deploys the original, and a document it refuses is outside the population (Section 6.3.1). Second, the command-line tool imports it and compiles the resulting source into a new document, the *rebuild*. A comparison procedure that ships with the test suite of BPMNscript, called the comparison below, then decides the relation of Section 4.5.4 between the model read from the original and the model read from the rebuild. Third, the engine deploys the rebuild.

Each model receives one of six outcomes (Table 6.1). Tolerances (i) and (iv) of the relation (Table 4.6) never make a model an exception, since they concern only the identifiers the compiler regenerates and the order of nodes and flows. A model needs one of the other eight tolerances if

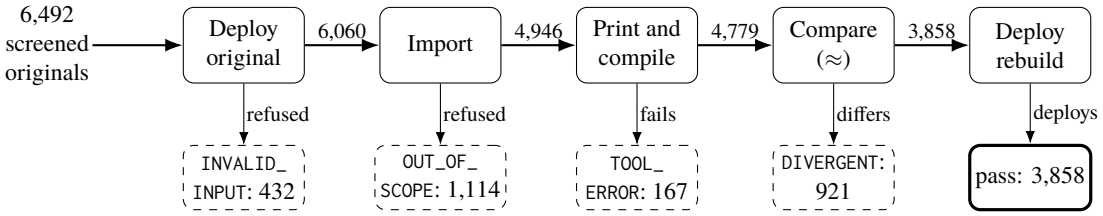


Figure 6.1: The steps of a round trip from a document and the outcome at each exit, with the counts of the mined corpus (Section 6.3.4). A pass is an equivalent pair whose rebuild deploys.

that tolerance belongs to a smallest set of tolerances whose removal makes the two models differ (Section 6.3.3). Such a set has two members where either tolerance alone makes the models equal, so a model can need several tolerances. The exception in `EQUIVALENT_BY_EXCEPTION`, a label the tool prints, stands for such a needed tolerance.

Table 6.1: The six outcomes of a round trip.

Outcome	Condition	Effect on the rates
`EQUIVALENT`	≈ holds, and no tolerance is needed	pass
`EQUIVALENT_BY_EXCEPTION`	≈ holds, and at least one tolerance is needed	pass
`DIVERGENT`	≈ does not hold	counts against
`OUT_OF_SCOPE`	the importer refuses the model with a typed error	bounds the claim
`TOOL_ERROR`	a crash, a timeout, or an output that the tool cannot read back, such as printed source that fails the validator	counts against
`INVALID_INPUT`	the engine refuses the original, an example fails to compile, or the validator rejects a generated program	outside the population

A round trip passes when the pair is `EQUIVALENT` or `EQUIVALENT_BY_EXCEPTION` and the rebuild deploys. For a generated program, only the second step applies, since the program comes from BPMNscript itself and the engine tests deploy what the compiler writes (Section 5.8). For an MIWG model, a pass is called conformance (Section 6.4), and the layout is out of scope (Section 4.5.3). Table 6.2 lists where the relation departs from the execution-relevant content that RQ4 asks about (Section 1.3). The relation compares names and documentation since they carry information for the developer (Section 4.2), and names are the labels of a model drawn before a developer adds the execution details (Section 1.1). It compares every condition on every flow by its syntax tree, including the leading `#` or `$`, so a condition that comes back as `${…}` from `#{…}` and a changed condition on a branch of a parallel gateway count as divergences, although the engine runs both alike. The discussion counts the divergent models with only such differences (Section 6.6). The last row is not measured, since the evaluation deploys the models but runs no instance. RQ4 is decided for each model: a model meets it if its round trip passes. For a set of models, the evaluation counts the models that pass and describes where the others differ. It gives their share both of the models that the importer accepts and of all models that the engine deploys.

The engine is the stock image of Operaton 2.1.0, pinned by digest, with the three mail libraries that the engine declares but does not ship, so that a mail task deploys. Two settings deviate

Table 6.2: Where the equivalence relation departs from execution-relevant content.

Direction	Content	Reported in
compared, without effect on execution	names and documentation; an expression written as `#{…}` instead of `${…}`; a condition on a branch of a parallel gateway, which Operaton ignores	divergences (Section 6.3.4)
dropped before the comparison, with effect on execution	Operaton settings without a field in the IR, such as an execution listener on a sequence flow	import warnings (Table 6.10)
tolerated, with effect on execution	which node the error of the engine reports when none of the conditions leaving a node holds (tolerance (viii))	needed tolerances (Table 6.7)
not compared	whether two models with equal fields run alike	–

from its defaults. The check `enforceHistoryTimeToLive` is off, so an original without a history time to live deploys, and the job executor is off, since a deployment is only checked and deleted again without starting an instance. Since no instance runs, the evaluation does not measure the performance of the engine, a quality property that workflow engine benchmarks address [HKL18, p. 152]. A deployment counts when the engine answers with success and creates at least one process definition, which the evaluation reads from the response of the engine's REST API, as workflow engine benchmarks query the deployment state through the API of the engine [HKL18, p. 159]. The engine refuses a document through a parse error, an XML error, or an unschedulable timer start event, for instance one whose timer or job priority is an expression without a value at deployment. Only these refusals make an original `INVALID_INPUT`, and any other failure of the engine stops the stage instead of becoming a data point.

6.2 Round Trips of Generated Programs

The stage runs the generator of the fuzz test of release 0.1.1 (Section 5.8) over the seeds 1 to 2,000. Before printing, the elements and flows of each imported model are put in three orderings: as generated, reversed, and shuffled with seed 1. Where the printer can choose between several structured forms, it decides by document order, so the orderings test whether its output depends on that order. The flows that leave one element keep their order among themselves. Each program is compiled, imported, printed, and compiled again, and the model of the first compilation is compared with the model of the second. Table 6.3 gives the outcome per ordering.

Table 6.3: Round trips of 2,000 generated programs per ordering of the imported model, by outcome.

Ordering	`EQUIVALENT`	`BY_EXCEPTION`	`DIVERGENT`	`INVALID`
as generated	1,662	334	0	4
reversed	1,645	347	4	4
shuffled	1,653	341	2	4

In every ordering, the validator rejects the same four programs, which are `INVALID_INPUT`, and the other 1,996 complete every step of the round trip without an error or a print warning (Section 5.5.2). Most of the tolerances they need are (v), which 328 programs need in each ordering, since the printer writes an expression in its own spelling. Tolerances (iii), pass-through gateways, (vii), generated end events, and (viii), flows entering or leaving a node directly, account for the few others. Reversed, 4 programs diverge, and shuffled, 2 of these 4 do. Each of them differs only in exclusive gateways without settings that the rebuild adds or removes.

6.3 Round-Trip Correctness on Real-World BPMN

This section measures the round trip on existing BPMN models. It selects a corpus of mined models (Section 6.3.1) and adds purposive sets from the repositories of Operaton, Camunda, and BPMNscript (Section 6.3.2). It then states how each pair is compared (Section 6.3.3) and reports the results of both stages (Section 6.3.4).

6.3.1 Corpus Selection

Population. The population consists of every executable BPMN 2.0 model that Operaton deploys, and an element or attribute is in scope if Operaton executes it (Section 4.2.1). Defining the population by the engine instead of by a list of supported elements keeps the evaluation from measuring the tool against its own scope. A model that Operaton refuses is `INVALID_INPUT`, is listed, and counts in no rate. In its default configuration, Operaton refuses a model without a history time to live (Section 4.3.5). The evaluation switches this check, `enforceHistoryTimeToLive`, off (Section 6.1), so the population includes such models.

Available datasets. The evaluation needs models written for an engine. Laue and Läuter [LL24, pp. 158–160] survey the public collections of BPMN models. Table 6.4 lists these collections and further candidates, each with the reason it is left out. Models mined from GitHub come from open-source software projects [TVH22b, p. 73]. Heinze et al. [HSA20, pp. 26–27] mined a random tenth of the repositories on GitHub, and Türker et al. [TVH22b, p. 69] extended the mining to all non-forked public repositories in the GHTorrent dump of March 2021. Their dataset is the largest public BPMN dataset known to them [TVH22b, p. 73], and the evaluation uses it as its sampling frame, the list of files from which the corpus is drawn.

The Federal Information Management (FIM) of Germany provides a method for turning legal requirements into digital public services [Hei25, p. 9]. Its XProzess module embeds BPMN 2.0 process models [Hei25, p. 9]. Its top layer describes services and processes as the law defines them, without the details of any authority that implements them [Hei25, p. 9]. In the nine FIM master processes we inspected, no process is marked executable, and none uses the Operaton or Camunda extensions, so none passes the screening below.

A new search of GitHub would add the repositories created after March 2021, but its search interface returns at most 1,000 results per query, so a complete search has to split its queries [BSG+23, pp. 63729–63730]. The evaluation keeps the published dataset as its only frame and leaves a new search to future work (Section 7.4.5).

Table 6.4: Public collections of BPMN models considered for the evaluation.

Collection	Models	Reason it is left out
SAP Signavio Academic Models [LL24, p. 158]	634,442	drawn by users of the modeling tool Signavio and stored in a proprietary JSON format
Camunda BPMN for Research [LL24, pp. 159, 166]	3,719	solutions of four training exercises, all of which also occur in GitHub repositories
RePROsitory [LL24, p. 159]	696	models digitized from the proceedings of the Business Process Management conference and its workshops
GenMyModel [LL24, p. 160]	26,123	drawn by users of an online modeling platform
GitHub crawl by Baalmann and Lübke [LL24, p. 159]	112,345	collection not public
BenchFlow [TVH22b, p. 73]	8,363	industrial models, 64 % of them BPMN, not public
National process platforms of Germany and Switzerland [LL24, p. 160]	2 and 152	both platforms discontinued
FIM master processes [Hei25, p. 9]	9	no process marked executable, no Operaton or Camunda extensions (see text)
GitHub mining of Türker et al. [TVH22b, p. 73]	79,713	used as the frame

Sampling frame. The frame is the dataset of Türker et al. [TVH22b], which lists BPMN files found in open-source repositories on GitHub. It provides only links to the models, because of their licenses [TVH22b, p. 70], and the link list is published under CC BY 4.0 [TVH22a]. The evaluation uses version 2.0.0 of the published dataset, pinned by its SHA-256 hash. It has 327,436 rows in 17,466 repositories, with 74,989 distinct contents, each identified by its Secure Hash Algorithm 1 (SHA-1) hash. Each row links one file by its repository, a branch of that repository, its path, and the SHA-1 of its content, and the frame records one branch for each repository. The 17,466 repositories, 327,436 rows, and 74,989 contents of the published file are fewer than the 18,534 repositories, 337,436 files, and 79,713 distinct models the paper reports [TVH22b, pp. 68, 70–71]. The evaluation counts what the published file holds. The 4,543 rows that refer to the empty file are left out, which leaves 74,988 non-empty contents in 17,415 repositories, since 51 repositories hold only such rows.

Acquisition and decay. Each repository is resolved to the current commit of its branch, and each content is fetched at that commit and checked against its SHA-1. A repository or branch counts as gone only if both git and the repository's web page on GitHub say so. Where the two disagree, the acquisition stops. The acquisition ran on October 17 and 18, 2026. Table 6.5 shows how much of the frame had decayed by then. Of the 17,415 repositories, 15,550 (89.3 %) resolve, 1,726 (9.9 %) are unavailable because the repository is deleted, private, or disabled by

GitHub, and 139 (0.8 %) have lost their branch. The evaluation repository publishes a list with the status of every content and every repository, but not the models, since they keep the licenses of their repositories.

Table 6.5: Status of the 74,988 non-empty distinct contents of the frame at acquisition.

Status	Contents	Share
intact (fetched, SHA-1 matches)	65,546	87.4 %
changed (fetched, SHA-1 differs)	823	1.1 %
missing (repository resolves, file absent)	1,792	2.4 %
unavailable (repository or branch gone)	6,827	9.1 %

Screening. Only intact contents are screened, and a changed content counts as decay, since it is no longer the model the frame recorded. A content passes screening if its XML parses, it holds a process with `isExecutable` set, and it uses the Operaton or Camunda extension namespace in at least one attribute or element. A namespace declaration alone does not count. Operaton 2.1.0 reads its extension attributes and elements from these two namespaces [Ope26a]. Before any parsing, a search of the text drops every content that lacks the text `isExecutable` or contains neither namespace URI, which removes 57,372 of the 65,546 intact contents. Of the remaining 8,174, 227 do not parse, 528 have no executable process, and 927 use neither extension, so 6,492 pass. Only these 6,492 reach the engine, which deploys 6,060 of them and refuses 432. The size of a model is the number of its flow nodes, that is, the activities, events, and gateways [OMG13, p. 97] of all its processes, including those inside sub-processes. The models fall into four size bands, fixed before the run: 1 to 10, 11 to 25, 26 to 50, and more than 50 flow nodes. The deployed ones form the population.

Deduplication. The frame holds many copies of the same file, and public BPMN repositories also hold models that differ in bytes but not in content. Such duplicates can distort the results of an analysis [LL24, pp. 162–163]. The evaluation therefore counts each distinct content once, by its SHA-1 hash, and every number below is in this count. Appendix G repeats the main results in two further counts: weighted by the copies of each content in the frame, and with contents grouped whose imported models are equal. The frame is used whole after filtering, which Baltes and Ralph [BR22, p. 5] call whole-frame sampling, so the rates describe this frame and claim nothing about BPMN models in general.

Specification check. The checker BPMNspector tests a model against the XML Schema, checks that the references of the model resolve, and checks most of the constraints that the schema does not express [GNV17, pp. 6–7]. About 23 % of the models of the frame pass it [TVH22b, p. 72], and BPMNspector 1.1.0 rejects 17 of the 26 golden files of BPMNscript (Section 5.8) [Kra26a]. Most of these fail on references that Operaton does not need. A message event of an executable process must carry an operation reference [OMG13, p. 270, Table 10.99], but Operaton correlates a message by its name [Ope26a]. The called element of a call activity must be a process of the

file or of an imported one [Gei13, pp. 16, 18], but Operaton looks the process up by its key at run time [Ope26a]. These two failures follow from how Operaton binds messages and called processes, so they affect any executable Operaton model with a message event or with a call activity to a process of another file. Five files contain an exclusive join reached by a single flow. The compiler keeps such a join where only one route of an `if` reaches it (Section 4.5.1). The specification forbids it [OMG13, p. 289, Sect. 10.6.1], but Operaton deploys and runs it [Ope26a]. A subset of specification-valid models would therefore drop the typical executable model, which is what RQ4 asks about, so the evaluation neither filters on BPMNspector nor reports it beside the outcomes.

6.3.2 Purposive Sets

A second stage runs the same round trip over models chosen on purpose and reports them per set, apart from the corpus. Table 6.12 lists the sets. All 2,502 BPMN files of the Operaton repository at commit `412a246` [Ope26a] are included: the 3 example processes, the 2,341 test models of its engine, and 158 further files, mostly test models of its database migrations and models of its web applications. The test models are written by the engine developers to exercise the parser and the runtime, so many are small and some are meant to be refused, which the engine sorts out as `INVALID_INPUT`. The 54 BPMN files of the repository `camunda-bpm-examples` at commit `6c7f4c4` [Cam26a] add the examples that Camunda maintains, since Operaton reads their namespace. The 32 example processes and the 26 golden files of BPMNscript complete the purposive sets. None of these models is real-world in the sense of the corpus, and some of them may also be in the frame.

6.3.3 Methodology and Tolerances

The comparison decides the relation of Section 4.5.4 in two stages. It first compares the models with expressions and generated event identifiers as text and without tolerance (vii), and if they differ, it compares the fully reduced models. For an equivalent pair, it also reports which tolerances the pair needed. Where two tolerances can each make the models equal on their own, the comparison reports the smallest groups whose joint removal makes them differ.

The comparison builds a single bijection between the nodes of the two models and matches nodes in document order where it cannot tell them apart by kind, neighbors, event definition, or, for a boundary event, the activity it is attached to. It can therefore report an equivalent pair as different, but never a different pair as equivalent, so the counts of equivalent models are lower bounds.

The relation and its tolerances are frozen with the release tag `v0.1.1` [Kra26b] before the evaluation is run. Tolerance (viii) counts conditioned flows that leave a node directly as the same as an inclusive fork after that node. If none of the conditions holds and there is no default flow, both models stop with an error, but the error identifies the fork in the model that has one and the node in the other (Appendix E). The reports count these pairs separately. Diagram data never reaches the relation and is counted, with three other kinds of loss, in an inventory per document: whether

the layout was regenerated, how many identifiers were replaced, how many gateway names were dropped, and how many of the original's element names the rebuild still carries. The printed source is measured apart from the outcome as well, since a model can round-trip to an equivalent model and still print with a `goto`.

6.3.4 Results

The results of the corpus come first, from the outcomes to the printed source, followed by those of the purposive sets.

Outcomes. Table 6.6 gives the outcomes of the 6,060 corpus models that Operaton deploys, each as a share of these deployed models. The importer accepts 4,946 of the 6,060 models and refuses the other 1,114 (18.4 %) with an error that identifies the construct. Of the 4,946 accepted models, 3,858 round-trip to an equivalent model, which is 78.0 % of the accepted and 63.7 % of the deployed models. Another 921 diverge, and for the remaining 167, the tool fails. The rebuild of every one of the 3,858 equivalent models deploys.

Table 6.6: Outcomes of the 6,060 corpus models that Operaton deploys, each with its share of the deployed models.

Outcome	Models	Share
`EQUIVALENT`	31	0.5 %
`EQUIVALENT_BY_EXCEPTION`	3,827	63.2 %
`DIVERGENT`	921	15.2 %
`OUT_OF_SCOPE`	1,114	18.4 %
`TOOL_ERROR`	167	2.8 %
population	6,060	

Needed tolerances. Table 6.7 shows which tolerances the equivalent models need. Tolerance (vi) dominates: 3,803 of the 3,827 models equivalent by exception lack a history time to live and come back with the default `P30D`, and 2,776 of them need no other tolerance. Of the 286 models that tolerance (viii) makes equivalent, 17 contain a node that leaves only through conditioned flows and has no default flow, where the error on a run without a matching condition identifies another node (Section 6.3.3).

Size. Most models are small, and the share of equivalent models falls with size (Table 6.8). Every equivalent model above ten flow nodes is `EQUIVALENT_BY_EXCEPTION`. Divergence and refusal together grow with size, from 20.8 % of the models with up to 10 flow nodes to 83.3 % of those with more than 50.

Table 6.7: Tolerances needed by the corpus models. A model can need several.

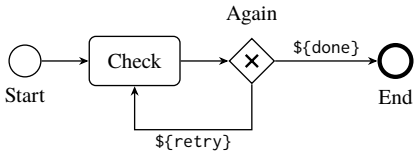
Tolerance	Models
(ii) gateway names	500
(iii) pass-through gateways	640
(v) expression syntax trees	356
(vi) default history time to live	3,803
(vii) generated end events	147
(viii) flows entering or leaving a node directly	286
of which an error on no match identifies the fork	17
(ix) unconditioned last flow	34
(x) nested parallel forks	3

Table 6.8: Outcomes of the 6,060 deployed corpus models by size band, with the share of equivalent models in each band.

Flow nodes	Models	Equivalent	Divergent	Refused	Tool error
1 to 10	4,237	3,276 (77.3 %)	388	493	80
11 to 25	1,280	501 (39.1 %)	305	426	48
26 to 50	441	75 (17.0 %)	192	146	28
more than 50	102	6 (5.9 %)	36	49	11
all	6,060	3,858 (63.7 %)	921	1,114	167

Divergences. In 787 of the 921 divergent models, every difference lies on an element that carries an import or print warning, and in the other 134, at least one does not (Section 6.7). Most divergences concern the routing. In 469 models, a condition expression differs. In 340 models the rebuild has extra sequence flows, and in 335 it misses flows of the original. In 268 models it adds an exclusive gateway, and in 186 it misses one. In 439 models, the comparison also reports that the outgoing flows of an exclusive gateway differ. In all of them, the gateways keep the order of their flows, and only a condition or a target differs. In 111 models, the default flow of an exclusive gateway differs, and beyond the routing, 95 differ in the declared error codes of the process. Every other difference occurs in fewer than 40 models.

Table 6.9 gives the causes of the condition differences, classified over all 469 models. The comparison keeps the leading `#` or `$` in front of the syntax tree it compares, so tolerance (v), expression syntax trees, does not accept a condition written as `#{…}` that comes back as `${…}`. For 248 divergent models, this rewrite is the only difference. Where the gateway that decides whether a loop repeats has two conditioned routes and no default flow, the printer recognizes the loop and drops the condition of the route that leaves it, so this exit becomes the default flow of the rebuild (Figure 6.2). In every one of these gateways, the exit is the second route. The importer wraps a bare-text condition without `${}`, such as `Si`, as an expression. For this reason, the rebuild does not deploy in 10 divergent models: Operaton cannot parse a wrapped text, such as `${some condition}`, at deployment [Ope26a].




```
process Review {
  var done: any
  var retry: any
  start Start
  do {
    step Check
  } while (retry)
  end End
}
```

Figure 6.2: A loop whose gateway `Again` has two conditioned routes and no default flow (left), and the source that the importer prints for it (right). The printer drops the condition `${done}` of the exit with a warning. If neither condition holds, Operaton stops the original with an error, and the rebuild leaves the loop.

Table 6.9: Causes of the differing conditions in the 469 divergent corpus models that differ in a condition expression. Models counts the models with at least one flow of that cause and Flows the differing flows; each flow has one cause, and a model can have several. Warning gives the step that reports the cause.

Cause	Models	Flows	Warning
#{…} written back as ${…}	295	651	import
condition of a loop exit dropped, the exit becomes the default flow	100	118	print
bare text wrapped as ${…}	56	224	import
condition dropped on the only route out of a task or event	13	43	print
condition of a loop exit dropped, the exit leads into a gateway	11	12	print
whitespace around an unparsed condition trimmed	9	15	none
condition dropped on a branch of a parallel gateway	3	6	print
decimal with a zero fraction written as an integer	2	5	none
condition dropped on a branch of an event-based gateway	1	2	print

Tool errors. All 167 tool errors arise at the same step: the importer accepts the model, but the printed source fails the validator of BPMNscript. We group the 406 errors that the validator reports in these models by the content they concern, and a model can fall under several groups. In 62 models, the validator rejects an expression or a setting, such as a condition that evaluates to a string instead of a Boolean, on which the engine throws, or an empty binding. In 43 models, the validator rejects the language of a script, FreeMarker in 37 of them. In 40 models, the validator rejects the control flow of the printed source, such as an unreachable statement after an `end` or a `goto`, or a `goto` that jumps into a branch. In 24 models, it rejects a form field, such as a field typed differently from an earlier declaration or a Boolean default other than `true` or `false`. In 5 models, the printed source does not parse or reuses a generated identifier.

Information loss. The round trip completes for 4,779 models, the accepted models without the 167 tool errors. Of these 4,779, 3,962 carry diagram data that is regenerated, 4,562 have at least one identifier replaced, and 1,013 lose at least one gateway name. Of the 32,128 element names of the originals, the rebuilds carry 23,749 (73.9 %). Most of the others are names that the DSL cannot carry, such as those of gateways, sequence flows, and boundary and intermediate events.

Table 6.10 counts the warnings of the importer over the 4,946 accepted models. Every model carries at least one warning for content rewritten into a form the engine reads alike, and 4,880 carry at least one for content the rebuild runs differently. Among these 4,880 are the 3,803 equivalent models that gain the default history time to live, which tolerance (vi) leaves out of the comparison (Section 4.5.4), and 4,797 of the 4,880 carry exactly one warning of this category. Other warnings of this category report, for example, a process imported as executable or a condition written as bare text and rewritten as an expression.

Table 6.10: Warnings of the importer over the 4,946 corpus models it accepts, by category: models with at least one warning, and warnings in total.

Category	Models	Warnings
content rewritten into a form the engine reads alike	4,946	6,429
content imported in a form the engine runs differently	4,880	5,243
name without a slot in the DSL	1,863	7,858
construct the importer does not read	1,122	6,773
Operaton setting without a field in the IR	1,088	3,602
unreferenced message or signal	422	896
lane	372	927
documentation without a slot in the DSL	43	76
construct carried as written that the engine misreads	24	30

Printed source. Of the 4,946 models the importer accepts, Table 6.11 counts those whose printed source contains a `goto` or a repair comment, which the printer writes above a split that it can print only as jumps (Listing 4.7) and in place of a jump it has to drop (Section 5.5.2). Over all accepted models, 20.9 % print with at least one of them, and the share with a `goto` grows with size. Models without a gateway also print a `goto`, mostly inside an event handler. An exclusive split that matches no pattern of the printer's catalogue gets no repair comment, since its chain of jumps keeps the full behavior (Section 4.5.2). No equivalent model carries a repair comment, and 12.5 % of the equivalent models contain a `goto`, against 41.9 % of the divergent ones. In a hand check of five small models with a `goto`, one contained a branch that the catalogue could have folded.

Table 6.11: Printed source of the 4,946 corpus models that the importer accepts, by size band: the share of models without any `goto` or repair comment, with at least one `goto`, and with at least one repair comment. A model can count under both `goto` and Repair comment.

Flow nodes	Models	Neither	`goto`	Repair comment
1 to 10	3,744	90.3 %	9.6 %	0.8 %
11 to 25	854	53.0 %	45.1 %	8.5 %
26 to 50	295	24.4 %	57.3 %	39.7 %
more than 50	53	15.1 %	75.5 %	28.3 %
all	4,946	79.1 %	19.3 %	4.8 %

Purposive sets. Table 6.12 gives the outcomes per purposive set. Over all purposive sets, of the 2,468 models that Operaton deploys, 2,072 (84.0 % of the deployed models) are equivalent, 136 (5.5 %) diverge, 260 (10.5 %) are refused, and none is a tool error. The official models of Operaton and Camunda are these sets without the examples and golden files of BPMNscript. Operaton deploys 2,411 of them, and the importer accepts 2,151 of these. Of the 2,151 accepted official models, 2,015 are equivalent, which is 93.7 % of the accepted and 83.6 % of the deployed official models. All 32 examples of BPMNscript are `EQUIVALENT`, and every rebuild of an equivalent model deploys. In 2 divergent models the rebuild does not deploy, for the same reason as in the corpus. The purposive sets print with fewer jumps than the corpus: of the 1,966 test models of Operaton that the importer accepts, 7.0 % contain a `goto` and 0.6 % a repair comment, against 19.3 % and 4.8 % of the accepted corpus models (Table 6.11).

Table 6.12: Outcomes per purposive set; `INVALID` counts the originals that the engine refuses or, for the examples and golden files, that do not compile.

Set	Files	EQUIVALENT	BY_EXCEPTION	DIVERGENT	OUT_OF_SCOPE	TOOL_ERROR	INVALID
BPMNscript examples	32	32	0	0	0	0	0
BPMNscript golden files	26	23	2	0	0	0	1
Operaton examples	3	1	1	1	0	0	0
Operaton engine tests	2,341	155	1,679	132	234	0	141
Operaton, other files	158	50	81	0	23	0	4
Camunda examples	54	44	4	3	3	0	0
total	2,614	305	1,767	136	260	0	146

6.4 BPMN MIWG Interchange Conformance

The MIWG publishes reference models that modeling tools import and export to test interchange [OMG26]. The evaluation uses all 22 of them, the reference models of the test suite at commit `386baa5` [MIWG26], exactly as published. Only 6 of them mark every process as executable, and 14 contain a collaboration. The stage therefore shows which parts of such drawings lie outside the scope of the DSL. It adds little evidence for RQ4, which concerns executable models. A model conforms when it imports without being refused, comes out `EQUIVALENT` or `EQUIVALENT_BY_EXCEPTION`, and its rebuild deploys. Whether the original deploys is recorded, but it does not gate the outcome.

Table 6.13 lists the outcome per model. Two models conform, A.1.0 and C.9.1. Both are equivalent, need only tolerance (vi), and their rebuilds deploy. The original of A.1.0 has a single process marked as not executable, so Operaton deploys no process definition from it. The importer reads such a process with a warning and writes it back as executable, so the rebuild of A.1.0 deploys one. The other 20 are refused with an error that identifies the construct, most often several processes of which none is executable (7 models) or a `bpmn:import` that is not Web Services Description Language (WSDL) (6). Operaton deploys no process definition from any of these 20 originals, so each refusal concerns a model the engine cannot run. Identifiers that are not valid bare identifiers

in the DSL, such as `WFP-6-` with its trailing hyphen, are printed in backticks (Section 4.3.4) and come back unchanged without a warning.

Table 6.13: Outcome of each of the 22 MIWG reference models.

Model	Outcome
A.1.0	conforms: `EQUIVALENT_BY_EXCEPTION` through (vi), rebuild deploys
A.2.0	refused: exclusive gateway with an unconditioned flow that is not its default
A.2.1	refused: condition in XPath
A.3.0	refused: sub-process without a start event
A.4.0, A.4.1	refused: several processes, none executable
B.1.0, B.2.0	refused: several processes, none executable
C.1.0	refused: service task without an execution binding
C.1.1	refused: `bpmn:import` that is not WSDL
C.2.0	refused: several processes, none executable
C.3.0	refused: sub-process without a start event
C.4.0, C.5.0	refused: several processes, none executable
C.6.0	refused: message without a name
C.7.0	refused: service task without an execution binding
C.8.0, C.8.1, C.9.0	refused: `bpmn:import` that is not WSDL
C.9.1	conforms: `EQUIVALENT_BY_EXCEPTION` through (vi), rebuild deploys
C.9.2, C.10.0	refused: `bpmn:import` that is not WSDL

6.5 Notation Size

RQ3 calls the DSL concise if a process takes fewer non-whitespace characters in it than in each related textual notation in which the same process is written (Section 1.3). Three processes have listings in related notations. We count the non-whitespace characters of each listing as printed, without its line numbers. The first is the invoice approval, which has listings in three related languages for executable BPMN (Section 3.11) and in kUML (Section 3.7). The DSL takes 246 characters for it, against 328 for Loadshift, 332 for the Workflow DSL, 573 for kUML, and 879 for the fluent builder. The second is the loan approval that the authors of BPELscript write in their language [BKLL09, Fig. 4]. Their full listing takes 1,262 characters. The excerpt without declarations and fault handler takes 751 (Listing 3.1). The DSL listing takes 411, its two variable declarations included (Listing 3.2). The third is the example workflow that the authors of the WEE DSL write in their language [MSS10, Listing 1]. Their full listing takes 911 characters. The excerpt without the handler, the endpoint and context declarations, and the two critical sections takes 518 (Listing 3.3 without lines 2–15, 20, 27, 30, and 37). The DSL listing takes 492, its two variable declarations included (Listing 3.4). Against the excerpt, the margin is small, since the DSL listing writes out the delegate class of each service task, where the WEE DSL refers to an endpoint by a short name. The excerpt in turn keeps the four blocks that store or add up the result of a call, which the DSL listing leaves to the delegate classes. Without the service bindings on both sides, that is, without the delegate classes and without the endpoints, arguments, and result blocks of the calls, the DSL listing takes 275 characters and the excerpt 320. The DSL is therefore concise on all three processes.

As a secondary measure, a stage of its own in the evaluation repository takes the 32 example processes of release 0.1.1, compiles each, removes the `bpmndi:BPMNDiagram` element, and counts the non-whitespace characters of the source, comments included, and of the XML. Every example is shorter in the DSL than in the XML. The XML takes a median of 6.8 times the characters of its source, from 3.4 times for the smallest ratio to 15.6 times for the largest. The ratio is smallest where a process consists mostly of settings, which both forms spell once, and largest where it consists of control flow, which the XML spells as gateways and sequence flows. The outage notice, one mail task with three fields and a comment of three lines, has a ratio of 3.4, and the parallel approval, one `parallel` block of two user tasks, has one of 15.6. One of the 32 examples, `loan-approval-kopp`, writes the block-structured loan approval of Kopp et al. [KMWL09, Fig. 4]. In it, three ratings run in parallel, a human assessor checks the request after a low internal rating, and a combined condition accepts or rejects the loan. Its source takes 1,206 characters and its XML 8,949. The engine tests run it on Operaton once with the manual assessment and once without it (Section 5.8).

An edit shows the same difference. We add `if (amount > 10000) { user CfoApproval(assignee: "cfo") }` before the end of the invoice-approval process. The source grows by 49 non-whitespace characters on 3 lines. The compiled XML without DI grows by 1,744 characters, and a line diff shows 23 lines added and 3 removed. The XML gains two exclusive gateways, a user task, and five sequence flows, and the sequence flow from the join of the existing `if` to the end event is replaced, since the identifier of a flow contains the identifiers of both of its ends. The two nodes that the replaced flow connected therefore change their references. Part of these characters are identifiers the compiler mints, and a person editing the XML by hand could choose shorter ones, but the elements and references stay.

6.6 Discussion

The discussion interprets the corpus results, sets them against the official models, and relates them to related work.

What the equivalence rests on. The 3,858 equivalent corpus models rest largely on tolerance (vi), the default history time to live, which 3,803 of them need (Table 6.7). These originals lack a setting that Operaton requires under its default configuration, so this tolerance says more about how the mined models were written than about the transformation. The history time to live is the age after which the history cleanup of the engine may delete the records of a finished instance (Section 4.3.5), so it does not affect how an instance runs and lies outside the execution-relevant content that RQ4 asks about (Section 1.3). If the outcome set tolerance (vi) aside as well, the 2,776 models that need only this tolerance would join the 31 `EQUIVALENT` ones, which gives 2,807 `EQUIVALENT` models, 56.8 % of the accepted and 46.3 % of the deployed ones. The other tolerances apply to between 3 and 640 models each (Table 6.7), and none of them alone carries the result.

Outcomes and execution relevance. Three counts relate the outcomes to the rows of Table 6.2. Of the 3,858 equivalent models, 755 carry an Operaton setting that the importer drops with a warning (second row). In 17 equivalent models, the error on a run without a matching condition identifies another node (third row). Of the 921 divergent models, 254 differ only in content without effect on execution (first row). In 248 of them, the only difference is a condition written as `#{…}` that comes back as `${…}`. In 3, it is documentation, and in 3, a condition on a branch of a parallel gateway, where Operaton ignores conditions [Ope26a]. These counts come from a filter over the published per-model results of the corpus [Kra26a].

Where the round trip diverges. Divergence concentrates in the conditions and in the gateways between the nodes. The most frequent condition difference, `#{…}` written back as `${…}` (Table 6.9), has no effect on execution, since the engine runs both forms alike [Ope26a]. The relation is thus stricter than the engine, and since it was frozen before the run (Section 6.3.3), the models whose only difference is this rewrite stay divergent. Of the condition differences that change behavior, most come from the printer, each time with a warning (Table 6.9). A dropped loop exit matters only in a run in which neither condition of the loop gateway holds, since the exit is the second route of each of these gateways. The original then stops with an error, and the rebuild leaves the loop. A dropped condition on the only route out of a task or event matters whenever that condition does not hold: the original stops with an error, and the rebuild continues [Ope26a]. A bare-text condition makes the original stop with an error whenever its route is evaluated, since Operaton evaluates it as text [Ope26a]. The rebuild evaluates the wrapped text as an expression. Sequence flows and exclusive gateways present on only one side are the next largest differences, and larger models offer more places for them, which is consistent with the share of equivalent models falling with size. Settings and task attributes, by contrast, rarely diverge: no field outside the routing differs in more than 95 models. Apart from the spelling of conditions, the XML reader and the compiler thus carry the content of a node through the round trip more reliably than the printer reconstructs the graph between the nodes. The generated programs point the same way, since those that diverge differ only in exclusive gateways, and only when the element order changes (Section 6.2). R1 confines `goto` to where an author writes one, but in imported source the printer decides where it stands. Of the accepted corpus models, 79.1 % print without a `goto` or repair comment, whereas of those above 50 flow nodes only 15.1 % do (Table 6.11). The printer writes a `goto` or a repair comment wherever its catalogue finds no block form. The 20.9 % of the accepted models that print with either are therefore an upper bound for the models that need one of them in the DSL. The bound is not tight, since the hand check found a branch that the catalogue could have folded (Section 6.3.4).

What the refusals mark. The 1,114 refused models show where the scope of the DSL meets models that are in use. The importer refuses them with a typed error that identifies the construct, and seven groups account for 952 of them, each listed with what the DSL writes:

- a connector (371 models): a service or send task calls an external service, such as a REST endpoint, through the connector plug-in of Operaton [Ope26a], and the DSL has no setting for it;
- several executable processes in one file (144): a DSL file holds one process;
- a message definition without a message name (119): the DSL refers to a message by its name;
- compensation through an association from a compensation boundary event to its handler (115): the DSL writes the handler as an `on compensation` block;
- a script that Operaton loads from an external resource (76): a script body in the DSL is inline;
- a call activity that starts a CMMN case (64): the DSL covers BPMN only;
- a condition that declares a language (63): conditions in the DSL are Java Unified Expression Language (JUEL) expressions (Section 4.3).

Apart from the CMMN case, each is a candidate for widening the scope (Section 7.4.5). The refusals of the MIWG models are of a different kind, since they concern models the engine cannot run (Section 6.4).

What the tool errors mark. The 167 tool errors are an inconsistency inside the tool: the importer accepts a model whose printed source the validator of the same tool rejects, for content such as an unsupported script language or a Boolean spelled in a way that the engine reads as false. The validator follows the engine more strictly than the importer here, and an import that refused or warned about these values would turn the tool errors into refusals or warnings.

Official and mined models. The official models of Operaton and Camunda round-trip better than the mined ones, and without a single tool error. Of the accepted models, 93.7 % of the official and 78.0 % of the mined ones are equivalent (Section 6.3.4). The official models are written by the developers of the engines to exercise its parser and runtime (Section 6.3.2), whereas the mined models come from open-source projects on GitHub. The purposive sets therefore test the tool against models written for the engine by its developers, and only the corpus tests it against models as they occur on GitHub.

Comparison with related work. Of the works in Table 3.2, the DSL is the only one that meets R1 to R3 (Section 3.12), and on models mined from GitHub, this evaluation measures R3 and, for R2, whether the rebuilt model deploys. On the invoice approval, the loan approval of BPELscript, and the example workflow of the WEE DSL, the DSL takes the fewest characters of the compared notations (Section 6.5). For R2, the rebuild of every equivalent model deploys on Operaton. The Workflow DSL meets R2 only partially: its authors report an executable export, but its tooling of January 2022 drops the condition of the invoice approval, and Operaton rejects the export (Section 3.12). For R3, the importer accepts 4,946 of the 6,060 mined models that Operaton deploys, and 3,858 of them round-trip to an equivalent model (Table 6.6). No other work in Table 3.2 describes an import of executable XML into source (Section 3.12). The closest precedent

for the round trip is StalkCD (Section 3.2), in which 717 of 1,027 Jenkinsfiles mined from GitHub (70 %) came back without information loss after nine normalization steps [DKH21, pp. 220–221]. The shares are of the same order: in this evaluation, the equivalent models are 63.7 % of the deployed mined models, and 70.4 % weighted by copies in the frame (Appendix G). The shares of StalkCD and of this evaluation are not directly comparable, however. StalkCD starts from the text: each Jenkinsfile is translated into the DSL of StalkCD and back, and the result counts as equal when it matches the original as text once formatting such as comments, whitespace, and the quotation marks of stage names is removed [DKH21, p. 221]. Its BPMN transformation is no part of this round trip. Here, every rebuild must also deploy on Operaton, and the relation also compares the names and documentation of the process, its activities, and its start and end events, so a round trip that loses one of these diverges (Section 4.5.4).

Where related work goes further. BPELscript covers every construct of BPEL [BKLL09, p. 39], whereas the DSL has a construct for 17 of the 19 element types that Operaton executes (Section 4.2.4). Bischof et al. [BKLL09, p. 45] also propose an edit script that keeps the details added to an earlier version, as long as its operations still apply, whereas the importer of the DSL regenerates the source (Section 3.10). Deriving the gateways also has a cost on import: an import into a notation that writes every gateway can map the graph element by element (Section 3.10), whereas the printer writes a `goto` in 19.3 % of the accepted mined models (Table 6.11).

6.7 Threats to Validity

Construct validity. The equivalence relation (Section 4.5.4) compares the fields of the IR, and the IR does not carry every setting the engine reads. The importer drops such a setting with a warning. Of the 4,946 corpus models that the importer accepts, 1,088 (22.0 %) carry such a warning (Table 6.10). A model with this warning can count as equivalent even where the dropped setting would change how its rebuild runs (Table 6.2). Structural equality also differs from equal behavior: the relation compares conditions by their syntax, and we do not prove that two models with equal fields run alike. A formal comparison of behavior would translate both models into Petri nets [DDO08], which this thesis does not do. The comparison belongs to the test suite of the tool it judges and shares the normalization of the model with it, so a defect common to both would go unnoticed. The generated programs, whose source is compiled twice by the same compiler, do not guard against this either. Finally, a warning does not cover every difference. A difference counts as covered when an import or print warning belongs to its element. Gateways, sequence flows, and boundary events are matched by their neighbors, ends, or host, so a warning on that neighbor, end, or host also covers a difference in them. In 134 of the 921 divergent corpus models (14.5 %), at least one difference lies on an element without a warning, so the round trip changes these models without notice. Of the purposive sets, only the test resources of Operaton contain such models, 58 of their 132 divergent ones. A warning is matched to a difference by the element alone, and it can concern another field of that element. The 787 corpus models whose

differences all carry a warning are therefore an upper bound for the models whose warnings explain the divergence.

Internal validity. Every number depends on the choices of deduplication, screening, and size bands, which are defined in the evaluation repository and reported with the corpus (Section 6.3.1). The engine configuration moves the boundary of the population: with `enforceHistoryTimeToLive` at its default, Operaton would refuse at least the 3,803 corpus models that lack a history time to live and need tolerance (vi). The evaluation code changed after the release tag `v0.1.1` and after a first run of the corpus stage, but no change touches the relation. Two changes affect the counts: a repository that GitHub shows as disabled counts as unavailable (four repositories), and an original with an unschedulable timer start event counts as `INVALID_INPUT` (13 originals). For 3 of these 13, the repetitions of the timer cycle all lie in the past, so their outcome depends on the date of the run. The other changes only add lines to the reports.

External validity. Public GitHub repositories are not the population of real-world BPMN models: more than two thirds of GitHub projects have a single committer, and many active projects also develop outside GitHub [KGB+16, Table 1]. The frame was collected in 2021 [TVH22a], before the Community Edition of Camunda 7 reached its end of life [Cam25], so it contains no models written since, and decay has removed 12.6 % of its distinct contents. Most mined models are small, which the size bands make visible, and the purposive sets are not real-world models at all. We wrote the 32 example processes whose size is compared with the XML. We also wrote the listings of the invoice approval in the three related languages and in kUML (Sections 3.7 and 3.11). The BPELscript listing of the loan approval and the WEE listing of the example workflow come from the authors of these languages [BKLL09, Fig. 4; MSS10, Listing 1], and we wrote their DSL listings.

Conclusion validity. All results are descriptive, and no statistical inference is drawn. The size bands and the screening criteria were set before the full run, and every reported count comes from one run of the final evaluation code over each input.

Chapter 7 answers the research questions from the results of this chapter and states the limitations of the language and of the evidence.

7 Conclusion and Future Work

This chapter draws together the design, implementation, and evaluation of Chapters 4 to 6. It summarizes the four contributions of Section 1.5 (Section 7.1) and answers the research questions (Section 7.2). It then states the limitations of the language and of the evidence (Section 7.3) and closes with directions for future work, most of which start from these limitations (Section 7.4).

7.1 Summary of Contributions

This thesis presents BPMNscript, a textual DSL for executable BPMN 2.0 processes on Operaton, with a compiler to BPMN 2.0 XML, an importer that translates such XML back into the DSL, and a language server for the editor. The syntax (Section 4.3) covers 17 of the 19 element types that Operaton executes and 52 of the 54 event combinations that the specification permits, all of which Operaton executes (Section 4.2.4). For the invoice approval, the loan approval of BPELscript, and the example workflow of the WEE DSL, it needs fewer characters than each related notation in which these processes are written (Section 6.5). The implementation (Chapter 5) compiles, imports, and supports editing with completion, navigation, rename, hover, and inline diagnostics, and the engine tests deploy its compiled output on Operaton (Section 5.8). The bidirectional transformation recovers structured statements from a gateway graph on import, and its equivalence relation judges every round trip of the evaluation (Section 4.5). Both are those of release 0.1.1 [Kra26b]. The evaluation (Chapter 6) measures the round trip on generated programs, official models, mined models, and the MIWG reference models. For the last three, it deploys each rebuilt model on Operaton: every rebuild of an equivalent model deploys, and the rebuilds of 12 divergent models, 10 mined and 2 official, fail to deploy (Section 6.3.4).

7.2 Answers to the Research Questions

The four research questions of Section 1.3 have the following answers.

RQ1. The textual process languages we found relate text and graphics in four ways (Section 3.12). Some replace the graphical model with code. Others keep a language and engine of their own that accept BPMN models as input. A third group generates BPMN from text in one direction, and the few that translate in both directions target BPEL, continuous integration and continuous delivery (CI/CD) pipelines, or engine-neutral diagrams, apart from the MontiCore Workflow DSL, whose

import from BPMN XML its authors state without describing it. None of the languages found meets the three requirements of Section 1.1. The nearest, the MontiCore Workflow DSL [DMM+22], misses R1, since it writes every gateway, and in our test Operaton rejects its export. RQ1 is thus answered within the notations we found, and among them the combination of the three requirements is new.

RQ2. Operaton executes 19 element types and all 54 event combinations that the specification permits (Table 4.1). The DSL has a construct for 17 of these element types (Section 4.2.4). The two element types without a construct are the manual task and the association. The manual task imports as a `step`, since Operaton runs it like a plain task. At runtime, the association links a compensation boundary event to its handler, and the DSL writes this handler as an `on compensation` block. Import also refuses a call activity that starts a CMMN case, since the DSL covers BPMN only. The DSL has a construct for 52 of the 54 event combinations, and import rewrites a none intermediate throw event into a `step` and refuses a compensation boundary event. On real models, the importer accepts most of what Operaton deploys: of the 6,060 mined models that Operaton deploys, it accepts 4,946 and refuses 1,114. Seven groups account for 952 of these refusals, led by connectors to external services (Section 6.6). For the element types and event combinations of Operaton 2.1.0, this answers RQ2: the DSL covers all but two element types and all but two event combinations, and import rewrites or refuses these four.

RQ3. The constructs of RQ2 are expressed with structured statements such as `if`, `while`, `parallel`, and `await`, from which the compiler synthesizes gateways and sequence flows, and with keyed settings in parentheses for what Operaton reads from an element (Section 4.3). Where no structured statement fits an imported graph, the printer, the part of import that writes source, falls back to `goto` (Section 4.5.2). Of the mined models that the importer accepts, 19.3 % contain at least one (Table 6.11). The engine tests deploy every compiled example on Operaton, as written and again after a round trip (Section 5.8). For the invoice approval, the loan approval of BPELscript, and the example workflow of the WEE DSL, the DSL listing takes fewer non-whitespace characters than every related listing, so the notation is concise in the sense of RQ3 on these processes (Sections 3.11 and 6.5). Over the 32 example processes, the XML without DI takes a median of 6.8 times the non-whitespace characters of the source. Every example process is also shorter in the DSL than in its XML. RQ3 is answered for the constructs of RQ2: the DSL is the most concise notation on the three processes with related listings, and shorter than the XML it compiles to on every example process.

RQ4. The round trip preserves the execution-relevant content, as the relation of Section 4.5.4 decides it, for each of the 1,996 valid generated programs with its elements in the generated order (Section 6.2). When the elements and sequence flows are reversed, this holds for all but 4 of them, and when they are shuffled, for all but 2. For real models, the share of models that pass is lower. Operaton deploys 6,060 mined models, and the importer accepts 4,946 of them. Of these 4,946 models, 3,858 (78.0 % of the accepted models) round-trip to an equivalent model whose

rebuild deploys. These 3,858 models are 63.7 % of the deployed mined models. In its default configuration, Operaton refuses a model without a history time to live, so the compiler adds the default value to every model without one (Section 4.3.5). For 3,803 of the 3,858 equivalent mined models, the original lacks this value, and the rebuild gains the default. These originals deploy only because the evaluation switches this check off (Section 6.3.1). The history time to live only controls when the engine may delete the records of finished instances, so it lies outside the execution-relevant content of RQ4 (Section 6.6). Of the 2,151 official models that the importer accepts, 2,015 (93.7 % of the accepted official models) round-trip to an equivalent model whose rebuild deploys. Every official model that the importer accepts prints source that compiles with the tool. Among the 921 mined models that diverge, 254 differ only in content without effect on execution. In 248 of them, a condition written as `#{…}` comes back as `${…}`, which Operaton evaluates alike [Ope26a]. Most other divergences lie in the conditions and gateways that the printer rebuilds (Section 6.6). Among the 6,060 deployed mined models, the share of equivalent models falls with size, from 77.3 % of models with up to 10 flow nodes to 5.9 % of those with more than 50. For 167 mined models, the printed source fails the validator of the tool (Sections 5.2.3 and 6.3.4). Of the 22 MIWG reference models, 2 conform, and the importer refuses the other 20, from none of which Operaton deploys a process definition (Section 6.4). RQ4 is answered as a share that falls with size: the round trip preserves the execution-relevant content of most small mined models, of few large ones, and of most official models.

7.3 Limitations

Scope of the language. The non-goals of Section 4.1 bound what the DSL offers. The layout is regenerated on every compilation, so a diagram arranged by hand is lost in a round trip. Import also drops lanes, pools, and the names of gateways, sequence flows, and boundary and intermediate events, each with a warning. A model drawn in a modeling tool thus keeps its process logic and loses these parts of its drawing (Section 6.3.4). The DSL targets Operaton alone, writes one process per file, and has no form for collaborations, choreographies, or conversations.

Structured source from arbitrary graphs. The printer folds a gateway graph into blocks only where an entry of its pattern catalogue applies, and every other edge becomes a `goto` (Section 4.5.2). A parallel, inclusive, or event-based split that degrades to jumps loses the information whether one, several, or the first-resolving route runs. The printer marks such a region with a repair comment, and the source compiles to a plain exclusive choice until a person repairs it by hand. Of the mined models that the importer accepts, 19.3 % print at least one `goto` and 4.8 % a repair comment, most of these models above ten flow nodes (Table 6.11). The catalogue is incomplete: in a hand check of five small models with a `goto` (Section 6.3.4), one contained a branch that the catalogue could have folded. A printer built on the Refined Process Structure Tree (RPST) (Section 2.3) would limit `goto` to rigid fragments (Section 7.4.6).

What the evaluation shows. The relation compares fields of the IR (Section 4.4), and the IR does not carry every setting the engine reads. An Operaton setting without a field is dropped with a warning, which 22.0 % of the accepted mined models carry, so a model can count as equivalent although its rebuild runs differently (Section 6.7). The relation is also syntactic: it compares conditions as written, and we do not prove that equivalent models behave alike. In 134 of the 921 divergent mined models, at least one changed element carries no warning, so only a comparison such as that of the evaluation reveals the change (Section 6.7). For the 167 mined models whose printed source fails the validator, the importer accepts content that the validator of the same tool refuses, and these imports need repair by hand (Section 7.4.5).

Basis of the evidence. The comparison belongs to the test suite of the tool it judges, so a defect common to both would go unnoticed (Section 6.7). The mined models come from public GitHub repositories collected in 2021 [TVH22a], and the official models are written by the developers of the engines to exercise their parser and runtime (Section 6.3.2), so neither stands for executable BPMN models in general. The classification of the elements and event combinations rests on our reading of the parser source of Operaton 2.1.0 and holds for that version (Section 4.2.1). We wrote the 32 examples on which the notation is measured, the listings of the invoice approval in the three related languages and in kUML, and the DSL listings of the loan approval and of the WEE example workflow (Sections 3.1, 3.3, 3.11 and 6.5). The assumption of Section 1.1, that statements from programming languages suit the targeted developers better than the diagram, rests on reading experiments with other notations (Section 2.2), and the choice among the six candidate syntaxes rests on an analytical comparison (Section 4.3.2).

7.4 Future Work

The first direction uses the DSL as the target of LLM-based generation (Section 7.4.1). Four directions extend what the language and its transformation cover: further engines, collaborations, a preserved diagram layout, and more importable models (Sections 7.4.2 to 7.4.5). The last two concern the printer and the tooling around the language (Sections 7.4.6 and 7.4.7).

7.4.1 AI-Assisted BPMN Authoring with Compiler Feedback

In the comparison of five LLM-based BPMN generation tools by Drakopoulos et al. [DMN+26, pp. 7, 11–12], correctness is the weakest quality dimension on average (Section 3.9). Two of its criteria, connected elements and splits followed by matching joins, are checks a compiler performs, so an LLM could iterate on its diagnostics. Another criterion, that the model represents the described process logic, covers gaps between the prompt and the output, such as a missing step. Detecting such gaps needs the prompt, whereas a compiler sees only the output. We leave open whether generating the DSL with such compiler feedback gives more consistent results than generating BPMN 2.0 XML directly. A DSL file also carries no DI elements and holds only

process logic and execution semantics. On the 32 example processes, the XML without DI takes a median of 6.8 times the non-whitespace characters of the source (Section 6.5), which may make the DSL a better generation target.

7.4.2 Multi-Engine Portability

The DSL targets Operaton alone (Section 4.1), and its IR models the semantics that Operaton executes (Section 4.4). A second engine would therefore need fields of its own in the IR and its own serialization of the extension attributes it reads. The compiler would also have to refuse, with its reason, every binding of a source that the selected engine lacks, as the importer refuses what the DSL cannot write (Section 4.5.2). Only the common subset of the engines would carry over unchanged, and for three open-source engines that subset covered 30 of 70 tested features [GHL+15, p. 28].

7.4.3 Collaboration, Pools, and Lanes

A DSL file holds one process (Section 4.1). Import keeps the process of a single pool and reports the pool with a warning, since Operaton reads a pool only for the diagram. Lanes are reported the same way (Table 6.10), since Operaton also reads them only for the diagram (Table 4.1). Operaton deploys each executable process of a file as a definition of its own [Ope26a]. A DSL file with several processes, or an import that writes one file per process, would remove the reason for which 144 mined models are refused (Section 6.6). Message flows between these processes would stay diagram data, since Operaton ignores them and correlates a message by its name (Table 4.1, Section 4.3.4).

7.4.4 Bidirectional Diagram Interchange Preservation

The compiler lays out the diagram on every compilation, and import discards the DI data, so a layout arranged by hand is lost in a round trip (Section 4.5.3). A separate file that maps identifiers to coordinates could still restore the shapes of elements with an authored name and lay out only the rest.

7.4.5 Wider Import Coverage

Most refusals of mined models fall into seven groups (Section 6.6), and a setting or statement for a group would remove one reason for refusal. Connectors are the largest group, with 371 models, so a setting for connectors on service and send tasks would reach the most models. After the 144 models with several processes in one file (Section 7.4.3), message definitions without a message name (119) and compensation through an association to its handler (115) follow. The importer could also refuse the content that the validator rejects in the 167 models, so that each import either

fails with its reason or prints source that passes the validator (Section 6.3.4). A new search of GitHub for repositories created after March 2021 would extend the corpus to newer Camunda 7 and Operaton models (Section 6.3.1).

7.4.6 Complete Structuring of Arbitrary Graphs

A printer built on the RPST would print every fragment except the rigid ones as a block and so confine `goto` to rigid fragments, a bound the pattern catalogue lacks (Section 4.5.2). Inside a rigid fragment, the method of Polyvyanyy et al. [PGD12, pp. 519, 536–537] could remove further jumps, but it assumes a sound, acyclic fragment without inclusive joins, conditions that imported models need not meet (Section 3.10). Structuring the rigid fragments of arbitrary graphs, or bounding the jumps left in them, therefore remains open.

7.4.7 Test Generation, Formatting, and Embedded Script Tooling

Three smaller directions concern the tooling around the language. Generating test cases for compiled processes would let a process definition ship with executable checks of its own control flow. Such a generator reads the compiled XML and needs no change to the language, so it can follow as separate work. A formatter would give the chosen syntax, variant B with parentheses and braces, the consistent indentation that variant D, the off-side rule, makes part of its syntax (Section 4.3.2). An embedded language server for script fences would offer completion inside a script body in its own language, where release 0.1.1 only highlights it (Section 5.3.1).

A Event Combinations

Section 4.2.1 counts the combinations of an event type and a position of the event. There are eleven event types: the none event and ten types with an event definition, such as message and timer. Table A.1 lists the ten positions. This gives 110 combinations.

Table A.1: The ten positions of an event counted in Section 4.2.1.

Group	Position
start	start of a process
	start of an interrupting event sub-process
	start of a non-interrupting event sub-process
	start of an embedded sub-process
in the flow	intermediate catch event in the flow
	intermediate catch event after an event-based gateway
	intermediate throw event
	interrupting boundary event
	non-interrupting boundary event
end	end event

The two multiple forms, in which one event carries several definitions, are left out because the deployment parser of Operaton reads only one definition per event [Ope26a]. Compensation is triggered only after the work it undoes has completed, so it does not interrupt the process [OMG13, Table 10.86], and the attributes that separate interrupting from non-interrupting events do not apply to it [OMG13, Tables 10.87, 10.92]. The specification thus does not settle in which of the two kinds of position a compensation event is permitted. We count it as permitted in the interrupting positions and as not permitted in the non-interrupting ones, since a start event is interrupting by default (`isInterrupting` is true) [OMG13, Table 10.87]. The specification permits 54 of the 110 combinations [OMG13, Tables 10.84–10.86, 10.88–10.90, 10.92, Sect. 10.6.6], and Operaton executes all 54 [Ope26a]. Operaton runs some of them only in a given place [Ope26a]. A cancel boundary event must sit on a transaction, and a cancel end event must lie directly inside one. An escalation boundary event must sit on a sub-process, a call activity, or a user task. A compensation event sub-process must lie inside an embedded sub-process. A message throw or end event sends its message only if it carries a binding, as a service task does, and runs as a none event otherwise [Ope26a]. Operaton also deploys four combinations that the specification does not permit: a non-interrupting compensation event sub-process and non-interrupting error, cancel, and compensation boundary events [Ope26a]. Of the other 52 combinations that the specification does not permit, 35 fail the deployment and 17 deploy with their type ignored.

Table A.2 gives the outcome of each combination. The Operaton classifications come from reading the deployment parser of Operaton 2.1.0 [Ope26a]. The import results come from running the import of release 0.1.1 [Kra26b] on one minimal executable document per combination. Import carries 52 of the 54 permitted combinations into a DSL construct. It rewrites the none intermediate throw event into a `step` with a warning and refuses the compensation boundary event (Section 4.2.4). It also refuses every combination the specification does not permit.

Table A.2: Outcome of every combination of an event type and a position. Each cell gives the Operaton 2.1.0 classification (e = executed, i = deployed with the type ignored, r = deployment fails) and the import result (T = carried into a DSL construct, R = rewritten with a warning, X = refused). A cell in parentheses is a combination the specification does not permit [OMG13, Tables 10.84–10.86, 10.88–10.90, 10.92, Sect. 10.6.6]. A permitted combination carries its BPMN symbol [OMG13, Table 10.93, pp. 259–260]. ESP = event sub-process; int. = interrupting; NI = non-interrupting.

	Start				Catch			Boundary		
Definition	process	ESP int.	ESP NI	embedded	in flow	after gateway	Throw	int.	NI	End
none	e T	(r X)	(r X)	e T	(r X)	(r X)	e R	(r X)	(r X)	e T
message	e T	e T	e T	(r X)	e T	e T	e T	e T	e T	e T
timer	e T	e T	e T	(r X)	e T	e T	(i X)	e T	e T	(i X)
conditional	e T	e T	e T	(r X)	e T	e T	(i X)	e T	e T	(i X)
signal	e T	e T	e T	(r X)	e T	e T	e T	e T	e T	e T
error	(i X)	e T	(r X)	(r X)	(r X)	(r X)	(i X)	e T	(e X)	e T
escalation	(i X)	e T	e T	(r X)	(r X)	(r X)	e T	e T	e T	e T
compensation	(i X)	e T	(e X)	(r X)	(r X)	(r X)	e T	e X	(e X)	e T
cancel	(i X)	(r X)	(r X)	(i X)	(r X)	(r X)	(i X)	e T	(e X)	e T
terminate	(i X)	(r X)	(r X)	(i X)	(r X)	(r X)	(i X)	(r X)	(r X)	e T
link	(i X)	(r X)	(r X)	(i X)	e T	(r X)	e T	(r X)	(r X)	(i X)

B Element Types

This appendix lists the 36 element types that Section 4.2.1 counts, with their outcome in Operaton and in import. An element type is a class of the BPMN metamodel [OMG13]. The count covers the process, every class that a process or collaboration diagram draws as a shape, a connector, or an activity marker, the documentation, and the DI data. The five event classes are left out, because Appendix A counts events by combination. Apart from the documentation, every setting that an element holds or refers to, such as an event definition, a message, or a resource role or property of a user task, counts as part of that element.

Table B.1 gives the outcome of each element type. The Operaton classifications come from reading the deployment parser of Operaton 2.1.0 [Ope26a]. The import column gives the behavior of release 0.1.1 [Kra26b].

Table B.1: Outcome of every element type except events. Operaton 2.1.0: exec = deployed with a runtime effect; read = read without runtime effect; ign.* = ignored with a parser warning, after which any sequence flow to or from the element fails the deployment; n.p. = not parsed. Import: T = carried into a DSL construct, R = rewritten with a warning, W = dropped with a warning, X = refused.

Element type	Operaton	Import	DSL construct
Process and activities			
`process`	exec	T; X if ≥ 2 executable	`process P {}`
`task`	exec	T	`step`
`userTask`	exec	T	`user`
`serviceTask`	exec (binding required)	T; X without binding	`service`
`sendTask`	exec (binding required)	T; X without binding	`send`
`receiveTask`	exec	T; X on a dangling message reference	`receive`
`scriptTask`	exec	T	`script` with fenced body
`businessRuleTask`	exec (binding required)	T; X without binding	`decide`
`manualTask`	exec, as a plain task	R (as `step`)	–
`subProcess`	exec	T	`subprocess`; event sub-process: `on` without host
`transaction`	exec	T	`attempt`
`adHocSubProcess`	ign.*	X	–
`callActivity`	exec	T; X for a CMMN case (Section 4.2.4)	`call` (process only)
`multiInstanceLoopCharacteristics`	exec	T	`for` clause
`standardLoopCharacteristics`	n.p.	W	–
Gateways and flows			
`exclusiveGateway`	exec	T	`if`, `while`, `do`
`parallelGateway`	exec	T	`parallel`
`inclusiveGateway`	exec	T	`parallel` with conditioned branches
`eventBasedGateway`	exec	T	`await { … }`
`complexGateway`	ign.*	X	–
`sequenceFlow` (plain, conditional, default)	exec	T	statement order, `goto`
Collaboration			
`participant` (pool)	read	W	–
`lane`	read	W	–
`messageFlow`	n.p.	W	–
Artifacts			
`association`	exec as link from a compensation boundary event to its handler; else read	X as link to a compensation handler; else W	–
`textAnnotation`	n.p.	W	–
`group`	n.p.	W	–
Data			
`dataObject`	n.p.	W	–
`dataObjectReference`	n.p.	W	–
`dataStoreReference`	n.p.	W	–
`dataInput`	n.p.	W	–
`dataOutput`	n.p.	W	–
`dataInputAssociation`	n.p.	W	–
`dataOutputAssociation`	n.p.	W	–
Documentation and diagram			
`documentation`	read	T; W where the DSL has no slot	`documentation` setting
`BPMNDiagram` (diagram interchange)	read	dropped without warning	regenerated

C Compiled Running Example

This appendix shows the BPMN XML compiled from Listing 1.1 by BPMNscript 0.1.1 [Kra26b]. The DI part, 110 lines of coordinates generated by `bpmn-auto-layout` (Section 4.5.3), is omitted. The identifiers of the gateways and flows are synthesized by the compiler (Section 4.3.4).

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
  ↪ xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
  ↪ xmlns:dc="http://www.omg.org/spec/DD/20100524/DC" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
  ↪ xmlns:operaton="http://operaton.org/schema/1.0/bpmn" id="Definitions_expense-claim"
  ↪ targetNamespace="http://bpmnscript.io/processes" exporter="BPMNscript" exporterVersion="0.1.1">
  <bpmn:process id="expense-claim" name="Expense Claim" isExecutable="true" operaton:historyTimeToLive="P30D">
    <bpmn:startEvent id="ClaimSubmitted">
      <bpmn:extensionElements>
        <operaton:formData>
          <operaton:formField id="amount" label="Claimed amount" type="long" />
        </operaton:formData>
      </bpmn:extensionElements>
      <bpmn:outgoing>Flow_ClaimSubmitted_CheckPolicy</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:serviceTask id="CheckPolicy" name="Check the expense policy"
    ↪   operaton:class="com.example.expense.CheckPolicyDelegate">
      <bpmn:incoming>Flow_ClaimSubmitted_CheckPolicy</bpmn:incoming>
      <bpmn:outgoing>Flow_CheckPolicy_Gateway_expense-claim_2_loop</bpmn:outgoing>
    </bpmn:serviceTask>
    <bpmn:exclusiveGateway id="Gateway_expense-claim_2_loop" default="Flow_Gateway_expense-claim_2_loop_default">
      <bpmn:incoming>Flow_ReviseClaim_Gateway_expense-claim_2_loop</bpmn:incoming>
      <bpmn:incoming>Flow_CheckPolicy_Gateway_expense-claim_2_loop</bpmn:incoming>
      <bpmn:outgoing>Flow_Gateway_expense-claim_2_loop_ReviseClaim</bpmn:outgoing>
      <bpmn:outgoing>Flow_Gateway_expense-claim_2_loop_default</bpmn:outgoing>
    </bpmn:exclusiveGateway>
    <bpmn:userTask id="ReviseClaim" name="Revise the claim" operaton:assignee="demo">
      <bpmn:extensionElements>
        <operaton:formData>
          <operaton:formField id="amount" label="Corrected amount" type="long" />
          <operaton:formField id="compliant" label="Claim now complies" type="boolean" />
        </operaton:formData>
      </bpmn:extensionElements>
      <bpmn:incoming>Flow_Gateway_expense-claim_2_loop_ReviseClaim</bpmn:incoming>
      <bpmn:outgoing>Flow_ReviseClaim_Gateway_expense-claim_2_loop</bpmn:outgoing>
    </bpmn:userTask>
    <bpmn:exclusiveGateway id="Gateway_expense-claim_3_split"
    ↪   default="Flow_Gateway_expense-claim_3_split_default">
      <bpmn:incoming>Flow_Gateway_expense-claim_2_loop_default</bpmn:incoming>
      <bpmn:outgoing>Flow_Gateway_expense-claim_3_split_ApproveClaim</bpmn:outgoing>
      <bpmn:outgoing>Flow_Gateway_expense-claim_3_split_default</bpmn:outgoing>
    </bpmn:exclusiveGateway>
    <bpmn:exclusiveGateway id="Gateway_expense-claim_3_join">
      <bpmn:incoming>Flow_ApproveClaim_Gateway_expense-claim_3_join</bpmn:incoming>
      <bpmn:incoming>Flow_AutoApprove_Gateway_expense-claim_3_join</bpmn:incoming>
      <bpmn:outgoing>Flow_Gateway_expense-claim_3_join_Gateway_expense-claim_4_fork</bpmn:outgoing>
    </bpmn:exclusiveGateway>
```

```
42      <bpmn:userTask id="ApproveClaim" name="Approve the claim" operaton:assignee="manager">
43        <bpmn:incoming>Flow_Gateway_expense-claim_3_split_ApproveClaim</bpmn:incoming>
44        <bpmn:outgoing>Flow_ApproveClaim_Gateway_expense-claim_3_join</bpmn:outgoing>
45      </bpmn:userTask>
46      <bpmn:serviceTask id="AutoApprove" name="Approve automatically"
        ↪  operaton:class="com.example.expense.AutoApproveDelegate">
47        <bpmn:incoming>Flow_Gateway_expense-claim_3_split_default</bpmn:incoming>
48        <bpmn:outgoing>Flow_AutoApprove_Gateway_expense-claim_3_join</bpmn:outgoing>
49      </bpmn:serviceTask>
50      <bpmn:parallelGateway id="Gateway_expense-claim_4_fork">
51        <bpmn:incoming>Flow_Gateway_expense-claim_3_join_Gateway_expense-claim_4_fork</bpmn:incoming>
52        <bpmn:outgoing>Flow_Gateway_expense-claim_4_fork_BookPayment</bpmn:outgoing>
53        <bpmn:outgoing>Flow_Gateway_expense-claim_4_fork_NotifyClaimant</bpmn:outgoing>
54      </bpmn:parallelGateway>
55      <bpmn:parallelGateway id="Gateway_expense-claim_4_join">
56        <bpmn:incoming>Flow_BookPayment_Gateway_expense-claim_4_join</bpmn:incoming>
57        <bpmn:incoming>Flow_NotifyClaimant_Gateway_expense-claim_4_join</bpmn:incoming>
58        <bpmn:outgoing>Flow_Gateway_expense-claim_4_join_ClaimSettled</bpmn:outgoing>
59      </bpmn:parallelGateway>
60      <bpmn:serviceTask id="BookPayment" name="Book the payment"
        ↪  operaton:class="com.example.expense.BookPaymentDelegate">
61        <bpmn:incoming>Flow_Gateway_expense-claim_4_fork_BookPayment</bpmn:incoming>
62        <bpmn:outgoing>Flow_BookPayment_Gateway_expense-claim_4_join</bpmn:outgoing>
63      </bpmn:serviceTask>
64      <bpmn:serviceTask id="NotifyClaimant" name="Notify the claimant"
        ↪  operaton:class="com.example.expense.NotifyDelegate">
65        <bpmn:incoming>Flow_Gateway_expense-claim_4_fork_NotifyClaimant</bpmn:incoming>
66        <bpmn:outgoing>Flow_NotifyClaimant_Gateway_expense-claim_4_join</bpmn:outgoing>
67      </bpmn:serviceTask>
68      <bpmn:endEvent id="ClaimSettled">
69        <bpmn:incoming>Flow_Gateway_expense-claim_4_join_ClaimSettled</bpmn:incoming>
70      </bpmn:endEvent>
71      <bpmn:sequenceFlow id="Flow_ClaimSubmitted_CheckPolicy" sourceRef="ClaimSubmitted" targetRef="CheckPolicy" />
72      <bpmn:sequenceFlow id="Flow_Gateway_expense-claim_2_loop_ReviseClaim" name="!compliant"
        ↪  sourceRef="Gateway_expense-claim_2_loop" targetRef="ReviseClaim">
73        <bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">${!compliant}</bpmn:conditionExpression>
74      </bpmn:sequenceFlow>
75      <bpmn:sequenceFlow id="Flow_ReviseClaim_Gateway_expense-claim_2_loop" sourceRef="ReviseClaim"
        ↪  targetRef="Gateway_expense-claim_2_loop" />
76      <bpmn:sequenceFlow id="Flow_CheckPolicy_Gateway_expense-claim_2_loop" sourceRef="CheckPolicy"
        ↪  targetRef="Gateway_expense-claim_2_loop" />
77      <bpmn:sequenceFlow id="Flow_Gateway_expense-claim_3_split_ApproveClaim" name="amount &#62; 500"
        ↪  sourceRef="Gateway_expense-claim_3_split" targetRef="ApproveClaim">
78        <bpmn:conditionExpression xsi:type="bpmn:tFormalExpression">${amount &gt; 500}</bpmn:conditionExpression>
79      </bpmn:sequenceFlow>
80      <bpmn:sequenceFlow id="Flow_ApproveClaim_Gateway_expense-claim_3_join" sourceRef="ApproveClaim"
        ↪  targetRef="Gateway_expense-claim_3_join" />
81      <bpmn:sequenceFlow id="Flow_Gateway_expense-claim_3_split_default" sourceRef="Gateway_expense-claim_3_split"
        ↪  targetRef="AutoApprove" />
82      <bpmn:sequenceFlow id="Flow_AutoApprove_Gateway_expense-claim_3_join" sourceRef="AutoApprove"
        ↪  targetRef="Gateway_expense-claim_3_join" />
83      <bpmn:sequenceFlow id="Flow_Gateway_expense-claim_2_loop_default" sourceRef="Gateway_expense-claim_2_loop"
        ↪  targetRef="Gateway_expense-claim_3_split" />
84      <bpmn:sequenceFlow id="Flow_Gateway_expense-claim_4_fork_BookPayment" sourceRef="Gateway_expense-claim_4_fork"
        ↪  targetRef="BookPayment" />
85      <bpmn:sequenceFlow id="Flow_BookPayment_Gateway_expense-claim_4_join" sourceRef="BookPayment"
        ↪  targetRef="Gateway_expense-claim_4_join" />
86      <bpmn:sequenceFlow id="Flow_Gateway_expense-claim_4_fork_NotifyClaimant"
        ↪  sourceRef="Gateway_expense-claim_4_fork" targetRef="NotifyClaimant" />
87      <bpmn:sequenceFlow id="Flow_NotifyClaimant_Gateway_expense-claim_4_join" sourceRef="NotifyClaimant"
        ↪  targetRef="Gateway_expense-claim_4_join" />
88      <bpmn:sequenceFlow id="Flow_Gateway_expense-claim_3_join_Gateway_expense-claim_4_fork"
        ↪  sourceRef="Gateway_expense-claim_3_join" targetRef="Gateway_expense-claim_4_fork" />
```

```
89      <bpmn:sequenceFlow id="Flow_Gateway_expense-claim_4_join_ClaimSettled"
        ↪  sourceRef="Gateway_expense-claim_4_join" targetRef="ClaimSettled" />
90    </bpmn:process>
91    <!-- bpmndi:BPMNDiagram (diagram interchange, 110 lines) omitted -->
92  </bpmn:definitions>
```

D Syntax Candidates

This appendix reproduces unchanged the six candidate syntaxes compared in Section 4.3.2, each a complete rendering of one order-handling process, which is larger than the excerpt in Listing 4.1. The line counts in Table 4.2 are the total number of lines of each listing, blank lines included. Variant B is the chosen syntax. Its listing is the draft and differs from the grammar of release 0.1.1 in the spelling of handlers, timer durations, external tasks, form field labels and defaults, and signal codes (Section 4.3.2).

List. D.1: Variant A, colon configuration: the `order-handling` process (71 lines).

```
process order-handling {
  label: "Order Handling"
  var amount: number
  var score: number
  error OUT_OF_STOCK {
    message: "Item unavailable"
  }

  start Placed {
    label: "Order placed"
    form {
      amount:   number { label: "Order amount" }
      priority: number { label: "Priority", default: 1 }
    }
  }

  user Review {
    label: "Review the order"
    assignee: "demo"
  }

  script Score {
    label: "Score the order"
    script: ```groovy
      def base = amount / priority
      def tier = base > 100 ? "high" : "low"
      execution.setVariable("score", base)
      execution.setVariable("tier", tier)
    ```
  }

  if (score < 50) {
    goto Review
  }
```
```

```
  service Charge {
    label: "Charge the customer"
    class: "com.example.ChargeDelegate"
  }

  external PrintLabel {
    label: "Print label"
    topic: "print-label"
  }

  if (amount > 1000) {
    user Approve { label: "Senior approval", assignee: "manager" }
  } else {
    service AutoApprove { label: "Auto-approve", delegate: "${approvalService}" }
  }

  subprocess Pack {
    label: "Pack the order"
    service Weigh { label: "Weigh parcel", class: "com.example.WeighDelegate" }
    external Handover { label: "Hand to carrier", topic: "carrier-handoff" }
  }

  emit signal Done { code: "OrderCompleted" }
  end Finished { label: "Order handled" }

  on Pack error OUT_OF_STOCK {
    catch: { code: c, message: m }
    user Contact { label: "Contact customer", assignee: "demo" }
  }

  on Review timer {
    after:     "PT2H"
    alongside: true
    service Remind { label: "Send reminder", class: "com.example.RemindDelegate" }
  }
}
```

List. D.2: Variant B, parentheses for attributes, braces for children: the `order-handling` process (53 lines).

```
process order-handling(label: "Order Handling") {
  var amount: number
  var score: number
  error OUT_OF_STOCK(message: "Item unavailable")

  start Placed(label: "Order placed") {
    form {
      amount:   number(label: "Order amount")
      priority: number(label: "Priority", default: 1)
    }
  }

  user Review(label: "Review the order", assignee: "demo")
```

```
  script Score(label: "Score the order") ```groovy
    def base = amount / priority
    def tier = base > 100 ? "high" : "low"
    execution.setVariable("score", base)
    execution.setVariable("tier", tier)
  ```

  if (score < 50) {
    goto Review
  }

  service Charge(label: "Charge the customer",
                 class: "com.example.ChargeDelegate")

  external PrintLabel(label: "Print label", topic: "print-label")

  if (amount > 1000) {
    user Approve(label: "Senior approval", assignee: "manager")
  } else {
    service AutoApprove(label: "Auto-approve", delegate: "${approvalService}")
  }

  subprocess Pack(label: "Pack the order") {
    service Weigh(label: "Weigh parcel", class: "com.example.WeighDelegate")
    external Handover(label: "Hand to carrier", topic: "carrier-handoff")
  }

  emit signal Done(code: "OrderCompleted")
  end Finished(label: "Order handled")

  on Pack error OUT_OF_STOCK(code: c, message: m) {
    user Contact(label: "Contact customer", assignee: "demo")
  }

  on Review timer(after: "PT2H", alongside) {
    service Remind(label: "Send reminder",
                   class: "com.example.RemindDelegate")
  }
}
```

**List. D.3:** Variant C, keyword prose: the `order-handling` process (53 lines).

```
process order-handling "Order Handling" {
  var amount is number
  var score is number
  error OUT_OF_STOCK means "Item unavailable"

  start Placed "Order placed" {
    form {
      amount   is number labeled "Order amount"
      priority is number labeled "Priority" default 1
    }
```
```

```
  }

  user Review "Review the order" assigned "demo"

  script Score "Score the order" ```groovy
    def base = amount / priority
    def tier = base > 100 ? "high" : "low"
    execution.setVariable("score", base)
    execution.setVariable("tier", tier)
  ```

  if score < 50 {
    goto Review
  }

  service Charge "Charge the customer"
    calls class "com.example.ChargeDelegate"

  external PrintLabel "Print label" on topic "print-label"

  if amount > 1000 {
    user Approve "Senior approval" assigned "manager"
  } else {
    service AutoApprove "Auto-approve" calls delegate "${approvalService}"
  }

  subprocess Pack "Pack the order" {
    service Weigh "Weigh parcel" calls class "com.example.WeighDelegate"
    external Handover "Hand to carrier" on topic "carrier-handoff"
  }

  emit signal Done "OrderCompleted"
  end Finished "Order handled"

  on Pack error OUT_OF_STOCK catching code c, message m {
    user Contact "Contact customer" assigned "demo"
  }

  on Review timer after "PT2H" alongside {
    service Remind "Send reminder"
      calls class "com.example.RemindDelegate"
  }
}
```

**List. D.4:** Variant D, off-side rule (indentation): the `order-handling` process (79 lines).

```
process order-handling:
  label: "Order Handling"
  var amount: number
  var score: number

  error OUT_OF_STOCK:
    message: "Item unavailable"
```
```

```
start Placed:
  label: "Order placed"
  form:
    amount:
      type: number
      label: "Order amount"
    priority:
      type: number
      label: "Priority"
      default: 1

user Review:
  label: "Review the order"
  assignee: "demo"

script Score:
  label: "Score the order"
  script: ```groovy
    def base = amount / priority
    def tier = base > 100 ? "high" : "low"
    execution.setVariable("score", base)
    execution.setVariable("tier", tier)
  ```

if score < 50:
  goto Review

service Charge:
  label: "Charge the customer"
  class: "com.example.ChargeDelegate"

external PrintLabel:
  label: "Print label"
  topic: "print-label"

if amount > 1000:
  user Approve:
    label: "Senior approval"
    assignee: "manager"
else:
  service AutoApprove:
    label: "Auto-approve"
    delegate: "${approvalService}"

subprocess Pack:
  label: "Pack the order"
  service Weigh:
    label: "Weigh parcel"
    class: "com.example.WeighDelegate"
  external Handover:
    label: "Hand to carrier"
    topic: "carrier-handoff"

emit signal Done:
```
```

```
      code: "OrderCompleted"

  end Finished:
    label: "Order handled"

  on Pack error OUT_OF_STOCK:
    catch: code c, message m
    user Contact:
      label: "Contact customer"
      assignee: "demo"

  on Review timer:
    after: "PT2H"
    alongside
    service Remind:
      label: "Send reminder"
      class: "com.example.RemindDelegate"
```

List. D.5: Variant E, decorator annotations: the `order-handling` process (88 lines).

```
process order-handling {
  @label "Order Handling"

  var amount: number
  var score: number

  @message "Item unavailable"
  error OUT_OF_STOCK

  @label "Order placed"
  start Placed {
    form {
      @label "Order amount"
      amount: number

      @label "Priority"
      @default 1
      priority: number
    }
  }

  @label "Review the order"
  @assignee "demo"
  user Review

  @label "Score the order"
  script Score ```groovy
    def base = amount / priority
    def tier = base > 100 ? "high" : "low"
    execution.setVariable("score", base)
    execution.setVariable("tier", tier)
  ```
```
```

```
  if score < 50 {
    goto Review
  }

  @label "Charge the customer"
  @class "com.example.ChargeDelegate"
  service Charge

  @label "Print label"
  @topic "print-label"
  external PrintLabel

  if amount > 1000 {
    @label "Senior approval"
    @assignee "manager"
    user Approve
  } else {
    @label "Auto-approve"
    @delegate "${approvalService}"
    service AutoApprove
  }

  @label "Pack the order"
  subprocess Pack {
    @label "Weigh parcel"
    @class "com.example.WeighDelegate"
    service Weigh

    @label "Hand to carrier"
    @topic "carrier-handoff"
    external Handover
  }

  @code "OrderCompleted"
  emit signal Done

  @label "Order handled"
  end Finished

  @catch code c
  @catch message m
  on Pack error OUT_OF_STOCK {
    @label "Contact customer"
    @assignee "demo"
    user Contact
  }

  @after "PT2H"
  @alongside
  on Review timer {
    @label "Send reminder"
    @class "com.example.RemindDelegate"
    service Remind
  }
}
```

List. D.6: Variant F, binding form: the `order-handling` process (51 lines).

```
process order-handling = process(label = "Order Handling") {
  amount = var(number)
  score  = var(number)

  OUT_OF_STOCK = error(message = "Item unavailable")

  Placed = start(label = "Order placed") {
    amount   = field(number, label = "Order amount")
    priority = field(number, label = "Priority", default = 1)
  }

  Review = user(label = "Review the order", assignee = "demo")

  Score = script(label = "Score the order") ```groovy
    def base = amount / priority
    def tier = base > 100 ? "high" : "low"
    execution.setVariable("score", base)
    execution.setVariable("tier", tier)
  ```

  if score < 50 {
    goto Review
  }

  Charge = service(label = "Charge the customer",
                   class = "com.example.ChargeDelegate")

  PrintLabel = external(label = "Print label", topic = "print-label")

  if amount > 1000 {
    Approve = user(label = "Senior approval", assignee = "manager")
  } else {
    AutoApprove = service(label = "Auto-approve", delegate = "${approvalService}")
  }

  Pack = subprocess(label = "Pack the order") {
    Weigh   = service(label = "Weigh parcel", class = "com.example.WeighDelegate")
    Handover = external(label = "Hand to carrier", topic = "carrier-handoff")
  }

  Done   = emit(signal, code = "OrderCompleted")
  Finished = end(label = "Order handled")

  on Pack error OUT_OF_STOCK (code = c, message = m) {
    Contact = user(label = "Contact customer", assignee = "demo")
  }

  on Review timer (after = "PT2H", alongside) {
    Remind = service(label = "Send reminder", class = "com.example.RemindDelegate")
  }
}
```
```

E Tolerances of the Equivalence Relation

This appendix gives the engine behavior behind each tolerance of the equivalence relation (Section 4.5.4), the exact conditions of tolerances (iii), (vii), (viii), and (x), and the order in which the reduction ρ applies all tolerances before two models are compared. That section also defines gateways without settings and authored identifiers. A flow container is the process, a sub-process or `attempt` block, or a handler without a host (Section 4.3.4).

Tolerance (i), authored identifiers. Only authored identifiers are compared. The others need no comparison, since the compiler regenerates them and the engine routes along the structure of the graph [Ope26a]. A replaced identifier no longer matches the activities in the history of earlier instances (Section 4.5.1), which the relation accepts because it compares process definitions only.

Tolerance (ii), gateway names. Gateway names are ignored, because the DSL has no slot for them and the printer reports each one it drops. Dropping them does not change routing, since BPMN defines the name of a flow element as its descriptive name [OMG13, Table 8.44], and Operaton routes without it [Ope26a].

Tolerance (iii), pass-through gateways. A gateway without settings that passes every token straight on is removed, and its incoming flows lead to its successor instead. This applies in two cases. It is removed if it is of any kind except the event-based one and has one incoming and one unconditioned outgoing flow. An exclusive merge, with several incoming flows and one unconditioned outgoing flow, is removed if the first node after it that the reduction keeps is an exclusive or event-based gateway. In front of a parallel or inclusive gateway, the merge stays, since that gateway synchronizes the flows the merge combined. A merge in front of any other node falls under tolerance (viii). A chain of such gateways is removed as a whole, and a cycle made only of them stays.

Tolerance (iv), order of nodes and flows. Nodes and flows are compared without regard to their order, except that the order of the non-default flows leaving an exclusive gateway is kept. Operaton takes the first of these flows in document order that carries no condition or one that holds [Ope26a], so two overlapping conditions in swapped order route differently, and so does an unconditioned flow moved in front of a conditioned one.

Tolerance (v), expression syntax trees. Expressions in five fields are compared by their syntax tree under the expression grammar of the DSL (Section 4.3.3): conditions, timers, code bindings, field injections, and multi-instance clauses. Whitespace thus does not count, and neither does

the choice between single and double quotes around a string literal, both of which Operaton's expression language accepts [Ope26a]. Parentheses do count, since the syntax tree keeps them. An expression in these fields outside that grammar is compared as text, and so is every expression in another field, such as an assignee or an input mapping.

Tolerance (vi), engine defaults. A missing history time to live counts as the default `P30D` that the compiler writes (Section 4.3.5), and the job flags `asyncBefore`, `asyncAfter`, and `exclusive` count as absent when they equal the engine default, since the IR stores them only when they differ from it. The XML reader applies this part of the tolerance when it builds the model, before the reduction runs.

Tolerance (vii), generated end events. A none end event with a generated identifier and without a name, documentation, or settings is removed if each flow into it is unconditioned and is the only outgoing flow of an activity or event. The compiler writes such an end event for a branch without `end`, and Operaton ends a token that finds no outgoing flow with the same call that a none end event makes [Ope26a]. The node in front of the end event may carry `asyncAfter`: without the end event, the end of that node becomes the asynchronous step, so both forms create the same jobs. A gateway is left out, since a gateway without outgoing flows runs its own code instead [Ope26a].

Tolerance (viii), flows entering or leaving a node directly. Several flows that enter or leave an activity or event directly count as the gateway that behaves the same way. An exclusive merge without settings in front of an activity or event is removed, since several flows into an activity behave as an exclusive gateway [OMG13, Sect. 13.3.1, p. 427]. Several outgoing flows of an activity behave as a parallel split, and as an inclusive split where they carry conditions [OMG13, Sect. 13.3.1, p. 427], and Operaton routes them in the same way (entry (2) in Appendix F). A parallel or inclusive fork without settings is therefore folded into the node in front of it if that node is not a gateway, its only outgoing flow leads to the fork and carries nothing but its two ends, and the fork has at least two outgoing flows, all of them unconditioned for a parallel fork. The default flow of an inclusive fork becomes the default flow of the node. The fork stays where the node routes differently [Ope26a]. With `asyncAfter` on the node or on its multi-instance clause, each outgoing flow becomes a job of its own. A node bound by `class` or `delegateExpression` may choose the outgoing flow itself. A multi-instance node in front of an inclusive fork with a default flow looks for the default flow among its own flows. If no condition holds and there is no default flow, both forms stop, but Operaton's error then identifies the fork instead of the node.

Tolerance (ix), unconditioned last flow. An unconditioned last flow of an exclusive gateway without a default flow counts as its default flow if every earlier outgoing flow is conditioned. Since Operaton takes the first such flow in document order (iv), this flow is taken exactly when a default flow would be. An inclusive gateway takes every unconditioned flow, so there such a flow is not a default [Ope26a].

Tolerance (x), nested parallel forks. A parallel fork without settings is merged into the parallel gateway in front of it if a single flow from that gateway enters it, this flow carries nothing but

its two ends, and the fork has at least two outgoing flows, all of them unconditioned. The outer gateway must not carry `asyncAfter`, which makes a job per outgoing flow [Ope26a], and a chain of nested forks merges into the outermost one. The printer writes such a nested fork for each join that synchronizes only some of the routes of a fork (Appendix F). A parallel gateway produces a token on every outgoing flow as soon as each incoming flow holds one [OMG13, Table 13.1], so the routes of both forks start together.

Order of the reduction. The reduction ρ applies the tolerances in three stages.

(1) On the whole model, it first applies the default history time to live (vi), then the comparison of expressions (v), and then the first part of (i), which replaces every generated identifier of an event or an event sub-process by a key derived from its kind and its event definition. Expressions are compared first, because this key contains the event definition.
(2) In every flow container, it applies, in this order: pass-through gateways (iii) together with the merge part of (viii), nested parallel forks (x), the fork part of (viii), unconditioned last flows (ix), and the second part of (i). This second part identifies every gateway, boundary event, event sub-process, and sequence flow by its neighbors in the graph, and it applies together with gateway names (ii) and the order of nodes and flows (iv).
(3) Last, it removes generated end events (vii).

F Restructuring Rules of the Printer

This appendix gives the full pattern catalogue of the printer and the rules by which it decides where a split continues (Section 4.5.2). The printer applies them to the flow graph of each flow container (the process, a sub-process or `attempt` block, or a handler without a host, Section 4.3.4), after computing dominators and post-dominators (Sections 2.3 and 5.5.2).

Passes before the catalogue. The first pass leaves out, with a warning, every element unreachable from any start event, boundary event, or event handler, since Operaton never runs it [Ope26a] and the DSL has no form for an unreachable activity. The *routes* of a split are the paths that start at its outgoing flows. Where a few routes of a parallel fork meet at a join of their own before the join of the fork, the second pass inserts an inner fork in front of these routes, so that the fork prints as nested `parallel` blocks. Every route still starts at once, each join waits for the same routes, and the printer reports the added fork.

The catalogue. The printer then applies a pattern catalogue whose eight entries fold seven patterns into blocks and turn every other edge into a jump:

(1) An exclusive split with a post-dominating join prints as `if`/`else if`/`else`.

(2) An activity or event with several outgoing flows prints as the fork that the engine runs it as, inclusive if any of the flows carries a condition or the element has a default flow, and parallel otherwise. This follows the engine, which leaves an activity by every outgoing flow whose condition holds or that carries none, and by its default flow only when no other flow is taken [Ope26a]. The one exception, a single conditioned flow beside the default flow, prints as `if`/`else`, which routes the same way.

(3) An unconditioned back edge into an exclusive gateway that dominates the body prints as `while`.

(4) A conditioned back edge from an exclusive gateway to a dominating body entry prints as `do` … `while`. Both loop entries apply only when the loop's route is the gateway's first route other than the default flow, since Operaton tests the routes of an exclusive gateway in document order [Ope26a].

(5) A parallel fork with a matching parallel join prints as `parallel`.

(6) An inclusive fork with a matching inclusive join prints as `parallel`, with a condition heading each conditioned branch.

(7) An event-based gateway whose every outgoing flow reaches an intermediate catch event prints as `await { … }`.

(8) Every other edge prints as `goto`, or, where the edge has no target that a jump could refer to, is dropped and marked by a repair comment (Section 4.5.2).

A fork without a join of its kind still prints as `parallel` in two cases: each of its routes ends on its own, or its routes meet at an element that only the fork's routes enter. In the second case, the element follows the block, and each branch ends in a `goto` to it. The compiler then generates no join for the block, so the element runs once per route, as in the model.

Where an exclusive split continues. For an exclusive split, the printer also has to decide where its branches end and printing continues. It tries six rules in a fixed order and uses the first that applies:

(a) The printer pairs splits with merges the way brackets are matched. If this pairing gives the split a merge of its own, the split continues at that merge.

(b) Let p be the nearest post-dominator of the split. If the split dominates p, and the first node of every route is dominated by the split and post-dominated by p, the split continues at p.

(c) The split continues at the first node that all of its continuing routes reach, preferring a merge. A route continues if it reaches the end of the enclosing body or, at the top level, the plain end event at the tail of the process, which the printer leaves out. This rule covers a split without a post-dominator inside the process because one of its branches stops at an end event of its own.

(d) If exactly one route carries no condition or starts with the default flow, the split continues at that route.

(e) If every route carries a condition and one of them stays inside the enclosing loop body or branch, the split continues at the end of that body or branch. The routes that leave that body or branch print as jumps.

(f) If the split has no default flow, every route carries a condition, and no enclosing loop body or branch offers an end to continue at, the split continues at the target of one route. A route qualifies if its target has a single incoming flow or has already been printed. The printer takes the last route if it qualifies, and otherwise the first qualifying route in document order. That route prints as an empty branch under its condition, and the chain closes with `else fail`, so that the recompiled split still stops the instance where no condition holds.

If no rule applies, every route prints as a jump.

Where a fork continues. Parallel and inclusive forks do not use these rules. A fork continues at its own merge, or else at the first merge of its kind that all of its continuing routes pass through. An event-based gateway does the same with an exclusive merge.

G Results in All Three Counts

The frame holds many copies of the same file (Section 6.3.1). Chapter 6 counts each distinct content once, by its SHA-1 hash. This appendix gives the main corpus results in two further counts beside it. The raw count weights each content by its copies in the frame. The structural count takes once each group of contents whose imported models are equal in the first stage of the comparison (Section 6.3.3). A group whose members end in different outcomes is split, and each member counts on its own, so the structural count never takes two different outcomes as one. Refused models have no imported model and count on their own, which leaves 5,931 units in the structural count.

Table G.1 gives the outcomes of the deployed corpus models in the three counts. In the raw count, the equivalent share of the deployed models is 70.4 %, higher than in the SHA-1 count, since the most copied models are small. The structural count differs from the SHA-1 count by less than one percentage point in every outcome. Table G.2 gives the needed tolerances in the three counts. In 366 of the 1,741 divergent models of the raw count, at least one difference lies on an element without a warning (Section 6.7).

Table G.1: Outcomes of the corpus models that Operaton deploys (6,060 distinct contents), in the raw, SHA-1, and structural count, each with its share of the deployed models in that count.

Outcome	Raw		SHA-1		Structural	
EQUIVALENT	104	0.7 %	31	0.5 %	31	0.5 %
EQUIVALENT_BY_EXCEPTION	10,721	69.8 %	3,827	63.2 %	3,735	63.0 %
DIVERGENT	1,741	11.3 %	921	15.2 %	887	15.0 %
OUT_OF_SCOPE	2,562	16.7 %	1,114	18.4 %	1,114	18.8 %
TOOL_ERROR	240	1.6 %	167	2.8 %	164	2.8 %
population	15,368		6,060		5,931	

Table G.2: Tolerances needed by the corpus models, in the raw, SHA-1, and structural count. A model can need several.

Tolerance	Raw	SHA-1	Structural
(ii) gateway names	876	500	484
(iii) pass-through gateways	1,060	640	626
(v) expression syntax trees	568	356	352
(vi) default history time to live	10,687	3,803	3,711
(vii) generated end events	544	147	146
(viii) flows entering or leaving a node directly	413	286	281
of which an error on no match identifies the fork	26	17	17
(ix) unconditioned last flow	48	34	33
(x) nested parallel forks	3	3	3

H Architectural Decision Records

Table H.1 lists the architectural decision records (ADRs) of BPMNscript with the sections of this thesis that each decision belongs to. They are published in the folder `docs/decisions` of release 0.1.1 [Kra26b], and each number links to its record.

Table H.1: The ADRs of BPMNscript at release 0.1.1, with the sections of this thesis that discuss each decision.

ADR	Title	Section
0000	Use Markdown Architectural Decision Records	4
0001	Use the Apache 2.0 license	5
0002	Use a mono-repo structure	5.1
0003	Use Langium as the language workbench	5.2
0004	Use VS Code as the primary IDE target	5.3
0005	In-editor conversion: a webview sidebar and two layers	5.6
0006	Target users without BPMN knowledge, and minimize boilerplate	4.1, 4.3.5
0007	An intermediate representation between the AST and BPMN XML	4.4
0008	Fork the Camunda moddle extension as a local Operaton one	5.4.3
0009	Diagram interchange: auto-layout and the sub-process expansion hint	4.5.3, 5.4.4
0010	Deterministic ids for synthesized elements, reserved, and authored ids kept as written	4.3.4, 4.5.1
0011	A structured, code-like grammar with one bracket shape	4.3.2
0012	Honest BPMN import: refusals, warnings, and the parse tiers	4.2.1
0013	The validator mirrors the engine's parse-time rules	4.1, 5.2.3
0014	Restructure the IR into a DSL with dominator analysis	4.5.2
0015	The print declares every variable it reads	4.2.4
0016	Task kinds on the authoring surface	4.3.3
0017	Service task bindings, including external, mail and shell	4.3.5
0018	Forms and field injection	4.3.5
0019	A call activity's variable mapping	4.3.5
0020	Listeners reuse `on` inside the attribute block	4.3.5
0021	Operaton engine attributes as named IR fields	4.3.5, 4.4
0022	Repetition: the `for` clause and its per-run job settings	4.3.4
0023	Event root elements derived from usage, and declared codes	4.3.4, 5.2.2
0024	Event trigger payloads in one paren slot	4.3.4
0025	Events at the process boundary: starts, ends and throws	4.2.1, 4.3.3
0026	Intermediate catch events	4.3.3
0027	Boundary events attached to an activity	4.3.3
0028	Compensation as a subprocess undo block	4.2.4
0029	Work that can be given up: `attempt` and `cancel`	4.3.3
0030	Link events for import round-trip symmetry	4.3.3
0031	Inclusive and event-based gateways	4.3.3, 4.5.1
0032	Expressions: raw templates, escapes, and what is lowered as JUEL	4.3.3, 5.4.1

I Use of AI Tools

Table I.1 lists the generative AI tools used for this thesis and its proposal, what each is used for, and where. Besides ScienceOS for the keyword search, the tools are Claude Code, the Claude desktop app, and the Claude models in GitHub Copilot, mostly with Claude Opus 4.6 to 5.5, Claude Fable 5.1 for larger reviews, and Claude Sonnet in a few cases. We revised and verified every passage produced or changed with these tools.

Table I.1: Generative AI tools, their use, and where they are used.

Used for	AI tool	Used in
Keyword search for literature	ScienceOS	Literature of all chapters (Section 1.4)
Ranking candidate works by relevance to the research questions; checking our reading of long works against their full text	Claude Opus	Literature of all chapters (Section 1.4)
Brainstorming the scope, the expected contributions, the chapter layout, and the timeline; checking the references; polishing the wording	Claude Code, Claude desktop app	Proposal
Checking the argument for logical consistency and readability, and cited claims against their sources	Claude Code	All chapters
Tracking the effect of an edit on other chapters, such as a result summarized in Chapter 1	Claude Code	All chapters
Detailed outline of chapters and sections	Claude Code	All chapters
Sorting our notes, decision records, code, meeting notes, and emails into the chapters as a starting point for writing	Claude Code	All chapters
Translating passages we wrote in German; checking grammar and spelling	Claude desktop app, GitHub Copilot, Claude Code	All chapters
Alternative structures and phrasings for paragraphs; drafts of paragraphs as a starting point for our own revision	Claude Code, Claude desktop app	All chapters, Chapter 5 to a larger extent, to match the code
LaTeX: listings, figures, TikZ, macros, and the consistent use of abbreviations	Claude Code	All chapters and appendices
Implementation and evaluation tools	GitHub Copilot, Claude Code	BPMNscript and the evaluation repository (Section 5.7)

Bibliography

[AAA+07] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, A. Yiu, eds. *Web Services Business Process Execution Language Version 2.0*. OASIS Standard. OASIS. Apr. 11, 2007. URL: https://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (visited on Sept. 26, 2026) (cit. on p. 16).

[Ada13] M. D. Adams. "Principled Parsing for Indentation-Sensitive Languages: Revisiting Landin's Offside Rule". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*. ACM, 2013, pp. 511–522. DOI: 10.1145/2429069.2429129 (cit. on p. 51).

[Airflow24] Apache Software Foundation. *What is Airflow?* Version 2.10.4. 2024. URL: https://airflow.apache.org/docs/apache-airflow/2.10.4/index.html (visited on Oct. 5, 2026) (cit. on p. 19).

[Bet26a] I. Betchvaia. *BPMN 2.0 DSL Reference*. kUML handbook, version 0.54.1, commit 8350d74. 2026. URL: https://github.com/kuml-dev/kUML/blob/8350d748d14324508bc33fe4a16413c0b86e2dcf/docs/handbook/modules/reference/pages/bpmn-dsl.adoc (visited on Oct. 27, 2026) (cit. on pp. 30, 31).

[Bet26b] I. Betchvaia. *kUML: Architecture That Compiles*. Version 0.54.1. Commit 8350d74. 2026. URL: https://github.com/kuml-dev/kUML/tree/v0.54.1 (visited on Oct. 9, 2026) (cit. on pp. 30, 31, 39, 40).

[Bet26c] I. Betchvaia. *ProcessBuilder.kt*. Version 0.54.1. kUML source, BPMN DSL builders, commit 8350d74. 2026. URL: https://github.com/kuml-dev/kUML/blob/8350d748d14324508bc33fe4a16413c0b86e2dcf/kuml-metamodel/kuml-metamodel-bpmn/src/commonMain/kotlin/dev/kuml/bpmn/dsl/ProcessBuilder.kt (visited on Oct. 27, 2026) (cit. on pp. 30, 31).

[BKLL09] M. Bischof, O. Kopp, T. van Lessen, F. Leymann. "BPELscript: A Simplified Script Syntax for WS-BPEL 2.0". In: *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009)*. IEEE, 2009, pp. 39–46. DOI: 10.1109/SEAA.2009.21 (cit. on pp. 3, 16, 21–23, 38, 97, 101, 102).

[bpmn.io26] bpmn.io. *bpmn-auto-layout*. Version 1.3.0. Mar. 11, 2026. URL: https://github.com/bpmn-io/bpmn-auto-layout/tree/v1.3.0 (visited on Oct. 10, 2026) (cit. on pp. 5, 64, 69, 78, 79).

[BR22] S. Baltes, P. Ralph. “Sampling in Software Engineering Research: A Critical Review and Guidelines”. In: *Empirical Software Engineering* 27.4, 94 (2022). DOI: 10.1007/s10664-021-10072-8 (cit. on p. 90).

[BSG+23] G. Buchgeher, S. Schöberl, V. Geist, B. Dorninger, P. Haindl, R. Weinreich. “Using Architecture Decision Records in Open Source Projects—An MSR Study on GitHub”. In: *IEEE Access* 11 (2023), pp. 63725–63740. DOI: 10.1109/ACCESS.2023.3287654 (cit. on p. 89).

[Bün19] H. Bünder. “Decoupling Language and Editor – The Impact of the Language Server Protocol on Textual Domain-Specific Languages”. In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*. SCITEPRESS, 2019, pp. 129–140. ISBN: 978-989-758-358-2. DOI: 10.5220/0007556301290140 (cit. on p. 18).

[Cam20a] Camunda. *BpmnYamlParser.java*. Version 0.25.0. Zeebe engine source. Nov. 6, 2020. URL: https://github.com/camunda/camunda/blob/0.25.0/engine/src/main/java/io/zeebe/engine/processing/deployment/model/yaml/BpmnYamlParser.java (visited on Sept. 26, 2026) (cit. on pp. 29, 38).

[Cam20b] Camunda. *YAML Workflows: Tasks and Control Flow*. Version 0.25.0. Zeebe documentation. Nov. 6, 2020. URL: https://github.com/camunda/camunda/tree/0.25.0/docs/src/yaml-workflows (visited on Sept. 26, 2026) (cit. on pp. 29, 38).

[Cam21a] Camunda. *Announcing Zeebe, Operate and Tasklist 1.0.0-alpha1*. Feb. 9, 2021. URL: https://camunda.com/blog/2021/02/announcing-zeebe-operate-and-tasklist-1-0-0-alpha1/ (visited on Mar. 3, 2026) (cit. on pp. 29, 38).

[Cam21b] Camunda. *Deprecated Features*. Zeebe documentation, section “Deprecated in 0.26.0”. Jan. 15, 2021. URL: https://github.com/camunda/camunda-docs/blob/19bf8335172b1056c4abe08afa5cd9ccba76fffa/docs/product-manuals/zeebe/open-source/deprecated-features.md (visited on Oct. 5, 2026) (cit. on p. 29).

[Cam25] Camunda. *Camunda 7 Enterprise End of Life (EoL) Extension Announcement*. Feb. 28, 2025. URL: https://camunda.com/blog/2025/02/camunda-7-enterprise-end-of-life-extension/ (visited on Sept. 26, 2026) (cit. on pp. 17, 102).

[Cam26a] Camunda. *camunda-bpm-examples*. Commit 6c7f4c4. 2026. URL: https://github.com/camunda/camunda-bpm-examples/tree/6c7f4c4adb4beb9a2f5d4c5e49fc1ddfc6dab3fc (visited on Oct. 27, 2026) (cit. on p. 91).

[Cam26b] Camunda. *Fluent Builder API*. Camunda 7 Manual, User Guide, Model API. 2026. URL: https://docs.camunda.org/manual/7.24/user-guide/model-api/bpmn-model-api/fluent-builder-api/ (visited on Oct. 8, 2026) (cit. on pp. 28, 34–38).

[CCFR24] I. Compagnucci, F. Corradini, F. Fornari, B. Re. "A Study on the Usage of the BPMN Notation for Designing Process Collaboration, Choreography, and Conversation Models". In: *Business & Information Systems Engineering* 66.1 (2024), pp. 43–66. ISSN: 1867-0202. DOI: 10.1007/s12599-023-00818-7 (cit. on p. 45).

[CFH+09] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, J. F. Terwilliger. "Bidirectional Transformations: A Cross-Discipline Perspective". In: *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT 2009)*. Vol. 5563. Lecture Notes in Computer Science. Springer, 2009, pp. 260–283. DOI: 10.1007/978-3-642-02408-5_19 (cit. on p. 33).

[CHK01] K. D. Cooper, T. J. Harvey, K. Kennedy. "A Simple, Fast Dominance Algorithm". Rice University. 2001. URL: https://www.cs.tufts.edu/comp/150FP/archive/keith-cooper/dom14.pdf (visited on Oct. 17, 2026) (cit. on p. 81).

[CHKH20] Y. Choi, N. L. Ha, P. Kongsuwan, K. H. Han. "An Alternative Method for Refined Process Structure Trees (RPST)". In: *Business Process Management Journal* 26.2 (2020), pp. 613–629. DOI: 10.1108/BPMJ-11-2018-0319 (cit. on pp. 15, 33, 64).

[CHR26] J. Charles, A. Hellwig, B. Rumpe. "Generating Language Servers for MontiCore-based DSLs". In: *Modellierung 2026*. Ed. by A. Koschmider, H.-G. Fill, A. Lepsien, F. Fonger. Lecture Notes in Informatics P-378. Bonn: Gesellschaft für Informatik e.V., 2026, pp. 123–139. DOI: 10.18420/modellierung2026-08 (cit. on pp. 18, 19, 70).

[CIB26] CIB software GmbH. *CIB seven: The Camunda 7 Alternative*. 2026. URL: https://cibseven.org/ (visited on Oct. 27, 2026) (cit. on p. 28).

[CTVW19] F. Ciccozzi, M. Tichy, H. Vangheluwe, D. Weyns. "Blended Modelling – What, Why and How". In: *Proceedings of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C 2019)*. IEEE, 2019, pp. 425–430. DOI: 10.1109/MODELS-C.2019.00068 (cit. on p. 31).

[DDO08] R. M. Dijkman, M. Dumas, C. Ouyang. "Semantics and Analysis of Business Process Models in BPMN". In: *Information and Software Technology* 50.12 (2008), pp. 1281–1294. DOI: 10.1016/j.infsof.2008.02.006 (cit. on p. 101).

[Dee24] N. Deehan. *Why BPMN Interchange Is So Important*. Camunda. Aug. 5, 2024. URL: https://camunda.com/blog/2024/08/why-bpmn-interchange-is-so-important/ (visited on Apr. 7, 2026) (cit. on p. 1).

[DKB08] G. Decker, O. Kopp, A. Barros. "An Introduction to Service Choreographies". In: *it – Information Technology* 50.2 (2008), pp. 122–127. DOI: 10.1524/itit.2008.0473 (cit. on p. 12).

[DKH21] T. F. Düllmann, O. Kabierschke, A. van Hoorn. “StalkCD: A Model-Driven Framework for Interoperability and Analysis of CI/CD Pipelines”. In: *Proceedings of the 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2021)*. IEEE, 2021, pp. 214–223. DOI: 10.1109/SEAA53835.2021.00035 (cit. on pp. 22, 38, 101).

[DLM+12] M. Dumas, M. La Rosa, J. Mendling, R. Mäesalu, H. A. Reijers, N. Semenenko. “Understanding Business Process Models: The Costs and Benefits of Structuredness”. In: *Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE 2012)*. Vol. 7328. Lecture Notes in Computer Science. Springer, 2012, pp. 31–46. DOI: 10.1007/978-3-642-31095-9_3 (cit. on p. 61).

[DLMR18] M. Dumas, M. La Rosa, J. Mendling, H. A. Reijers. *Fundamentals of Business Process Management*. 2nd ed. Springer, 2018. ISBN: 978-3-662-56508-7. DOI: 10.1007/978-3-662-56509-4 (cit. on pp. 1, 2).

[DMM+22] I. Drave, J. Michael, E. Müller, B. Rumpe, S. Varga. “Model-Driven Engineering of Process-Aware Information Systems”. In: *SN Computer Science* 3.6, 479 (2022). DOI: 10.1007/s42979-022-01334-3 (cit. on pp. 19, 27, 28, 33–35, 38, 40, 104).

[DMN+26] P. Drakopoulos, P. Malousoudis, N. Nousias, G. Tsakalidis, K. Vergidis. “Do LLMs Speak BPMN? An Evaluation of Their Process Modeling Capabilities Based on Quality Measures”. In: *Computation* 14.1, 10 (2026). ISSN: 2079-3197. DOI: 10.3390/computation14010010 (cit. on pp. 31, 106).

[Ecl26] Eclipse Foundation. *Xtext: Language Engineering for Everyone*. 2026. URL: https://eclipse.dev/Xtext/ (visited on Apr. 8, 2026) (cit. on p. 19).

[ESV+13] S. Erdweg, T. van der Storm, M. Voelter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, J. van der Woning. “The State of the Art in Language Workbenches. Conclusions from the Language Workbench Challenge”. In: *Proceedings of the 6th International Conference on Software Language Engineering (SLE 2013)*. Vol. 8225. Lecture Notes in Computer Science. Springer, 2013, pp. 197–217. DOI: 10.1007/978-3-319-02654-1_11 (cit. on p. 18).

[FGM+07] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt. “Combinators for Bidirectional Tree Transformations. A Linguistic Approach to the View-Update Problem”. In: *ACM Transactions on Programming Languages and Systems* 29.3, 17 (2007). DOI: 10.1145/1232420.1232424 (cit. on pp. 33, 66).

[FMP11] F. Friedrich, J. Mendling, F. Puhlmann. “Process Model Generation from Natural Language Text”. In: *Proceedings of the 23rd International Conference on Advanced Information Systems Engineering (CAiSE 2011)*. Vol. 6741. Lecture Notes in Computer Science. Springer, 2011, pp. 482–496. ISBN: 978-3-642-21639-8. DOI: 10.1007/978-3-642-21640-4_36 (cit. on p. 31).

[FPP08] J. N. Foster, A. Pilkiewicz, B. C. Pierce. “Quotient Lenses”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP ’08)*. ACM, 2008, pp. 383–396. DOI: 10.1145/1411204.1411257 (cit. on pp. 33, 67).

[Gei13] M. Geiger. *BPMN 2.0 Process Model Serialization Constraints*. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik 92. Otto-Friedrich-Universität Bamberg, 2013. URL: https://www.uni-bamberg.de/fileadmin/pi/Dateien/Publikationen/techrep-bpmn-serialization-constraints.pdf (visited on Oct. 23, 2026) (cit. on p. 91).

[GFM19] V. Garousi, M. Felderer, M. V. Mäntylä. “Guidelines for Including Grey Literature and Conducting Multivocal Literature Reviews in Software Engineering”. In: *Information and Software Technology* 106 (2019), pp. 101–121. DOI: 10.1016/j.infsof.2018.09.006 (cit. on p. 7).

[GH26] GitHub, Inc. *Workflow Syntax for GitHub Actions*. 2026. URL: https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax (visited on Oct. 8, 2026) (cit. on pp. 25, 38).

[GHA11] N. Genon, P. Heymans, D. Amyot. “Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation”. In: *Software Language Engineering: Third International Conference (SLE 2010)*. Vol. 6563. Lecture Notes in Computer Science. Springer, 2011, pp. 377–396. ISBN: 978-3-642-19439-9. DOI: 10.1007/978-3-642-19440-5_25 (cit. on p. 14).

[GHL+15] M. Geiger, S. Harrer, J. Lenhard, M. Casar, A. Vorndran, G. Wirtz. “BPMN Conformance in Open Source Engines”. In: *Proceedings of the 9th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2015)*. IEEE, 2015, pp. 21–30. DOI: 10.1109/SOSE.2015.22 (cit. on pp. 3, 16, 17, 107).

[GHLW18] M. Geiger, S. Harrer, J. Lenhard, G. Wirtz. “BPMN 2.0: The State of Support and Implementation”. In: *Future Generation Computer Systems* 80 (2018), pp. 250–262. DOI: 10.1016/j.future.2017.01.006 (cit. on pp. 1, 3, 16–18).

[GKR+07] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel. “Text-based Modeling”. In: *Proceedings of the 4th International Workshop on Software Language Engineering (ateM 2007)*. Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007. arXiv: 1409.6623 (cit. on pp. 2, 14).

[GNV17] M. Geiger, P. Neugebauer, A. Vorndran. “Automatic Standard Compliance Assessment of BPMN 2.0 Process Models”. In: *Proceedings of the 9th Central European Workshop on Services and their Composition (ZEUS 2017)*. Ed. by O. Kopp, J. Lenhard, C. Pautasso. Vol. 1826. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 4–10 (cit. on pp. 1, 90).

[Har17] S. Harrer. “Effective and Efficient Process Engine Evaluation”. PhD thesis. Otto-Friedrich-Universität Bamberg, 2017. DOI: 10.20378/irbo-49633 (cit. on pp. 1, 17).

[Hei25] T. S. Heinze. “Studying Domain Dependence in BPMN Process Modeling: An Empirical Research Proposal”. In: *Proceedings of the 17th Central European Workshop on Services and their Composition (ZEUS 2025)*. Ed. by S. Böhm, M. Vidgof. Vol. 4030. CEUR Workshop Proceedings. CEUR-WS.org, 2025, pp. 8–11 (cit. on pp. 88, 89).

[HK04] R. Hauser, J. Koehler. “Compiling Process Graphs into Executable Code”. In: *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE 2004)*. Ed. by G. Karsai, E. Visser. Vol. 3286. Lecture Notes in Computer Science. Springer, 2004, pp. 317–336. DOI: 10.1007/978-3-540-30175-2_17 (cit. on pp. 32, 33, 81).

[HKL18] S. Harrer, O. Kopp, J. Lenhard. “Patterns for Workflow Engine Benchmarking”. In: *Advances in Service-Oriented and Cloud Computing – Workshops of ESOCC 2016*. Vol. 707. Communications in Computer and Information Science. Springer, 2018, pp. 151–163. DOI: 10.1007/978-3-319-72125-5_12 (cit. on p. 87).

[HSA20] T. S. Heinze, V. Stefanko, W. Amme. “BPMN in the Wild: BPMN on GitHub.com”. In: *Proceedings of the 12th ZEUS Workshop on Services and their Composition (ZEUS 2020)*. Ed. by J. Manner, S. Haarmann, S. Kolb, O. Kopp. Vol. 2575. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 26–29 (cit. on p. 88).

[ISP20] A. Ivanchikj, S. Serbout, C. Pautasso. “From Text to Visual BPMN Process Models: Design and Evaluation”. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2020)*. ACM, 2020, pp. 229–239. DOI: 10.1145/3365438.3410990 (cit. on pp. 28, 29, 38, 39).

[ISP22] A. Ivanchikj, S. Serbout, C. Pautasso. “Live Process Modeling with the BPMN Sketch Miner”. In: *Software and Systems Modeling* 21.5 (2022), pp. 1877–1906. DOI: 10.1007/s10270-022-01009-w (cit. on pp. 1, 13, 29).

[KAZ18] O. Kopp, A. Armbruster, O. Zimmermann. “Markdown Architectural Decision Records: Format and Tool Support”. In: *Proceedings of the 10th Central European Workshop on Services and their Composition (ZEUS 2018)*. Ed. by N. Herzberg, C. Hochreiner, O. Kopp, J. Lenhard. Vol. 2072. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 55–62 (cit. on pp. 8, 41).

[KC07] B. Kitchenham, S. Charters. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. EBSE Technical Report EBSE-2007-01. Version 2.3. Keele University and University of Durham, July 9, 2007 (cit. on p. 7).

[KGB+16] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian. "An In-Depth Study of the Promises and Perils of Mining GitHub". In: *Empirical Software Engineering* 21.5 (2016), pp. 2035–2071. DOI: 10.1007/s10664-015-9393-5 (cit. on p. 102).

[KHB00] B. Kiepuszewski, A. H. M. ter Hofstede, C. J. Bussler. "On Structured Workflow Modelling". In: *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*. Ed. by B. Wangler, L. Bergman. Vol. 1789. Lecture Notes in Computer Science. Springer, 2000, pp. 431–445. DOI: 10.1007/3-540-45140-4_29 (cit. on pp. 3, 14, 15, 21).

[KHCR14] M. Kapuruge, J. Han, A. Colman, U. Rüegg. "EPClets: A Lightweight and Flexible Textual Language to Augment EPC Process Modelling". In: *2014 IEEE International Conference on Services Computing (SCC)*. IEEE, 2014, pp. 693–700. DOI: 10.1109/SCC.2014.96 (cit. on pp. 27, 38).

[KKP+09] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, S. Völkel. "Design Guidelines for Domain Specific Languages". In: *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*. Ed. by M. Rossi, J. Sprinkle, J. Gray, J.-P. Tolvanen. Technical Report B-108. Helsinki School of Economics, 2009, pp. 7–13. ISBN: 978-952-488-371-9. arXiv: 1409.2378 (cit. on pp. 18, 50, 51).

[KMWL09] O. Kopp, D. Martin, D. Wutke, F. Leymann. "The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages". In: *Enterprise Modelling and Information Systems Architectures* 4.1 (2009), pp. 3–13. DOI: 10.18417/emisa.4.1.1 (cit. on pp. 14, 16, 98).

[Kra26a] M. Kranz. *BPMNscript Evaluation*. Version 1.1.0. Commit 16ef457. Oct. 23, 2026. URL: https://github.com/BPMNscript/validation/tree/v1.1.0 (visited on Oct. 23, 2026) (cit. on pp. 85, 90, 99).

[Kra26b] M. Kranz. *BPMNscript: A Textual Language for Executable BPMN 2.0 Processes*. Version 0.1.1. Commit 4989a31. Oct. 17, 2026. URL: https://github.com/BPMNscript/bpmnscript/tree/v0.1.1 (visited on Oct. 17, 2026) (cit. on pp. 4, 8, 12, 34, 41, 47, 48, 69, 74, 85, 91, 103, 110, 111, 113, 133).

[Kra26c] M. Kranz. *devline: A Development Pipeline Plugin for Claude Code*. Version 0.4.4. 2026. URL: https://github.com/Conava/claude-devline (visited on Oct. 25, 2026) (cit. on p. 82).

[KSB24] N. Krieger, S. Speth, S. Becker. “HyLiMo: A Hybrid Live-Synchronized Modular Diagramming Editor as IDE Extension for Technical and Scientific Publications”. In: *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments (IDE 2024)*. ACM, 2024, pp. 70–75. DOI: 10.1145/3643796.3648458 (cit. on pp. 30, 31, 39).

[KWD+04] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, R. Valk. “An Extensible Editor and Simulation Engine for Petri Nets: Renew”. In: *Applications and Theory of Petri Nets 2004, 25th International Conference (ICATPN 2004)*. Vol. 3099. Lecture Notes in Computer Science. Springer, 2004, pp. 484–493. DOI: 10.1007/978-3-540-27793-4_29 (cit. on pp. 31, 39).

[Lan66] P. J. Landin. “The Next 700 Programming Languages”. In: *Communications of the ACM* 9.3 (1966), pp. 157–166. DOI: 10.1145/365230.365257 (cit. on p. 51).

[Langium26a] TypeFox GmbH. *Langium*. 2026. URL: https://langium.org/ (visited on Apr. 8, 2026) (cit. on pp. 19, 71).

[Langium26b] TypeFox GmbH. *Langium*. Version 4.4.0. Sept. 9, 2026. URL: https://github.com/eclipse-langium/langium/tree/v4.4.0 (visited on Oct. 27, 2026) (cit. on pp. 69, 70).

[Langium26c] TypeFox GmbH. *Langium Documentation: Configuration via Services*. 2026. URL: https://langium.org/docs/reference/configuration-services/ (visited on Oct. 17, 2026) (cit. on p. 71).

[Langium26d] TypeFox GmbH. *Langium Documentation: Document Lifecycle*. 2026. URL: https://langium.org/docs/reference/document-lifecycle/ (visited on Oct. 17, 2026) (cit. on pp. 71, 72).

[Langium26e] TypeFox GmbH. *Langium Documentation: Generate the AST*. 2026. URL: https://langium.org/docs/learn/workflow/generate_ast/ (visited on Oct. 9, 2026) (cit. on pp. 19, 71, 74).

[Langium26f] TypeFox GmbH. *Langium Documentation: Keywords as Identifiers*. 2026. URL: https://langium.org/docs/recipes/keywords-as-identifiers/ (visited on Oct. 27, 2026) (cit. on p. 71).

[Langium26g] TypeFox GmbH. *Langium Recipes: Indentation-Sensitive Languages*. 2026. URL: https://langium.org/docs/recipes/lexing/indentation-sensitive-languages/ (visited on Sept. 17, 2026) (cit. on p. 51).

[LC25] C. Lauwers, C. Curescu, eds. *TOSCA Version 2.0*. OASIS Standard. OASIS. July 22, 2025. URL: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html (visited on Apr. 8, 2026) (cit. on pp. 25, 38).

[Len16] J. Lenhard. “Portability of Process-Aware and Service-Oriented Software: Evidence and Metrics”. PhD thesis. Bamberg: Otto-Friedrich-Universität Bamberg, 2016. ISBN: 978-3-86309-399-0. DOI: 10.20378/irb-40174 (cit. on p. 17).

[LL24] R. Laue, M. Läuter. “Beobachtungen und Einsichten zu Repositorys von BPMN-Modellen”. In: *Modellierung 2024*. Ed. by M. Weske, J. Michael. Lecture Notes in Informatics P-348. Bonn: Gesellschaft für Informatik e.V., 2024, pp. 157–173. ISBN: 978-3-88579-742-5. DOI: 10.18420/modellierung2024_015 (cit. on pp. 88–90).

[MH11] S. Mazanek, M. Hanus. “Constructing a Bidirectional Transformation between BPMN and BPEL with a Functional Logic Programming Language”. In: *Journal of Visual Languages & Computing* 22.1 (2011), pp. 66–89. DOI: 10.1016/j.jvlc.2010.11.005 (cit. on p. 32).

[MHS05] M. Mernik, J. Heering, A. M. Sloane. “When and How to Develop Domain-Specific Languages”. In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892 (cit. on pp. 2, 18, 41, 51).

[Miragon26] Miragon GmbH. *bpmn-to-code: Gradle and Maven Plugin That Bridges Gaps Between BPMN and Code*. Version 5.2.0. Commit ee41a74. 2026. URL: https://github.com/Miragon/bpmn-to-code/tree/v5.2.0 (visited on Oct. 8, 2026) (cit. on p. 29).

[MIWG26] BPMN Model Interchange Working Group. *BPMN MIWG Test Suite, Reference Models*. Commit 386baa5. Oct. 7, 2026. URL: https://github.com/bpmn-miwg/bpmn-miwg-test-suite/tree/386baa55bf9a200f13ce26bef18557eceb19031f/Reference (visited on Oct. 19, 2026) (cit. on p. 96).

[MontiCore22] MontiCore Contributors. *Business Process Model and Notation (BPMN)*. Commit 6f4d86a. 2022. URL: https://github.com/MontiCore/bpmn/tree/6f4d86ab9a9a37ecc6705f045ffd7de063b34763 (visited on Oct. 21, 2026) (cit. on p. 40).

[MontiCore26] MontiCore Contributors. *Business Process Model and Notation (BPMN)*. Commit 0176e59. 2026. URL: https://github.com/MontiCore/bpmn/tree/0176e5976c713fff5fa5529f6c4e8d48c089c7ef (visited on Oct. 8, 2026) (cit. on pp. 27, 28, 33–35, 37, 38).

[MR14] J. Mangler, S. Rinderle-Ma. “CPEE – Cloud Process Execution Engine”. In: *Proceedings of the BPM Demo Sessions 2014*. Ed. by L. Limonad, B. Weber. Vol. 1295. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 51–55 (cit. on p. 23).

[MR22] J. Mangler, S. Rinderle-Ma. *Cloud Process Execution Engine: Architecture and Interfaces*. arXiv preprint. TU München, 2022. arXiv: 2208.12214 [cs.OH] (cit. on pp. 23, 25, 38).

[MSS10] J. Mangler, G. Stuermer, E. Schikuta. *Cloud Process Execution Engine – Evaluation of the Core Concepts*. arXiv preprint. University of Vienna, Workflow Systems and Technologies Group, 2010. arXiv: 1003.3330 [cs.SE] (cit. on pp. 23–26, 38, 97, 102).

[MU09] A.-H. Mogos, A. Urzica. “TN4PM: A Textual Notation for Process Modelling”. In: *Intelligent Distributed Computing III*. Ed. by G. A. Papadopoulos, C. Badica. Vol. 237. Studies in Computational Intelligence. Berlin, Heidelberg: Springer, 2009, pp. 263–268. DOI: 10.1007/978-3-642-03214-1_29 (cit. on pp. 25, 27).

[Nel26] G. Nelles. *loadshift-kotlin: Durable AI and Business Workflows in Kotlin, on Camunda 7, CIB seven and Camunda 8*. Commit d78881d. 2026. URL: https://github.com/it-atelier-gn/loadshift-kotlin/tree/d78881d995b6ab5bd57efe3f108ec9a3dd98797a (visited on Sept. 26, 2026) (cit. on pp. 28, 35–38).

[ODHA06] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, W. M. P. van der Aalst. “From BPMN Process Models to BPEL Web Services”. In: *Proceedings of the IEEE International Conference on Web Services (ICWS 2006)*. IEEE Computer Society, 2006, pp. 285–292. DOI: 10.1109/ICWS.2006.67 (cit. on pp. 15, 16).

[OFR+12] A. Ottensooser, A. Fekete, H. A. Reijers, J. Mendling, C. Menictas. “Making Sense of Business Process Descriptions: An Experimental Comparison of Graphical and Textual Notations”. In: *Journal of Systems and Software* 85.3 (2012), pp. 596–606. DOI: 10.1016/j.jss.2011.09.023 (cit. on pp. 2, 13).

[OMG13] Object Management Group. *Business Process Model and Notation (BPMN), Version 2.0.2*. OMG Document Number formal/2013-12-09. 2013. URL: https://www.omg.org/spec/BPMN/2.0.2/ (visited on Apr. 7, 2026) (cit. on pp. 1–3, 6, 9, 11–14, 16, 24, 25, 42–48, 52, 56, 59, 60, 90, 91, 109–111, 125–127).

[OMG16] Object Management Group. *Case Management Model and Notation (CMMN), Version 1.1*. OMG Document Number formal/2016-12-01. 2016. URL: https://www.omg.org/spec/CMMN/1.1/ (visited on Oct. 26, 2026) (cit. on p. 46).

[OMG26] Object Management Group. *BPMN Model Interchange Working Group (BPMN MIWG)*. 2026. URL: https://www.omgwiki.org/bpmn-miwg/doku.php (visited on Apr. 7, 2026) (cit. on pp. 1, 19, 85, 96).

[Ope25] Operaton Contributors. *Operaton: Open Source BPM Engine*. 2025. URL: https://operaton.org (visited on Mar. 3, 2026) (cit. on pp. 1, 3, 17).

[Ope26a] Operaton Contributors. *Operaton*. Version 2.1.0. Commit 412a246. Apr. 24, 2026. URL: https://github.com/operaton/operaton/tree/v2.1.0 (visited on Sept. 26, 2026) (cit. on pp. 3, 6, 17, 18, 28, 34–36, 43–46, 51, 55–57, 59, 61, 63, 73, 74, 79, 83, 90, 91, 93, 99, 100, 105, 107, 109–111, 125–127, 129).

[Ope26b] Operaton Contributors. *Operaton Releases*. 2026. URL: https://github.com/operaton/operaton/releases (visited on Sept. 26, 2026) (cit. on p. 17).

[Pet95] M. Petre. “Why Looking Isn’t Always Seeing: Readership Skills and Graphical Programming”. In: *Communications of the ACM* 38.6 (1995), pp. 33–44. DOI: 10.1145/203241.203251 (cit. on pp. 2, 13, 14).

[PGD12] A. Polyvyanyy, L. García-Bañuelos, M. Dumas. "Structuring Acyclic Process Models". In: *Information Systems* 37.6 (2012), pp. 518–538. DOI: 10.1016/j.is.2011.10.005 (cit. on pp. 3, 15, 33, 108).

[PlantBPMN26] Some1. *PlantBPMN: Converter from PlantUML Activity Diagrams to BPMN 2.0 XML Files*. Version 1.0.4. Commit f0c4ee3. 2026. URL: https://codeberg.org/Some1/PlantBPMN/src/tag/v1.0.4 (visited on Sept. 26, 2026) (cit. on pp. 28, 29, 39).

[PVV11] A. Polyvyanyy, J. Vanhatalo, H. Völzer. "Simplified Computation and Generalization of the Refined Process Structure Tree". In: *Web Services and Formal Methods (WS-FM 2010)*. Vol. 6551. Lecture Notes in Computer Science. Springer, 2011, pp. 25–41. ISBN: 978-3-642-19589-1. DOI: 10.1007/978-3-642-19589-1_2 (cit. on p. 15).

[Python26] Python Software Foundation. *The Python Language Reference, Release 3.13: Lexical Analysis*. 2026. URL: https://docs.python.org/3.13/reference/lexical_analysis.html#soft-keywords (visited on Oct. 24, 2026) (cit. on p. 53).

[Reh26a] N. Rehwaldt. *bpmn-moddle*. Version 10.3.1. Sept. 21, 2026. URL: https://github.com/bpmn-io/bpmn-moddle/tree/v10.3.1 (visited on Oct. 16, 2026) (cit. on pp. 69, 78–80, 83).

[Reh26b] N. Rehwaldt. *camunda-bpmn-moddle*. Version 8.0.2. Sept. 30, 2026. URL: https://github.com/camunda/camunda-bpmn-moddle/tree/v8.0.2 (visited on Oct. 16, 2026) (cit. on p. 78).

[RH17] B. Rumpe, K. Hölldobler. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering 32. Aachen: Shaker Verlag, 2017. ISBN: 978-3-8440-5713-3 (cit. on p. 19).

[SS13] A. Stefik, S. Siebert. "An Empirical Investigation into Programming Language Syntax". In: *ACM Transactions on Computing Education* 13.4, 19 (2013). DOI: 10.1145/2534973 (cit. on p. 51).

[SS25] M. Sujew, M. Spönemann. *Xtext, Langium, what next?* TypeFox. Dec. 16, 2025. URL: https://www.typefox.io/blog/xtext-langium-what-next/ (visited on Sept. 18, 2026) (cit. on pp. 18, 19).

[Temporal24] Temporal Technologies. *Temporal Workflow Definition*. 2024. URL: https://docs.temporal.io/workflow-definition (visited on Oct. 5, 2026) (cit. on p. 19).

[TVH22a] J. Türker, M. Völske, T. Heinze. *GitHub BPMN Artifacts Dataset 2021*. Version 2.0.0. Zenodo, Jan. 23, 2022. DOI: 10.5281/zenodo.5903352 (cit. on pp. 89, 102, 106).

[TVH22b] J. Türker, M. Völske, T. S. Heinze. “BPMN in the Wild: A Reprise”. In: *Proceedings of the 14th Central European Workshop on Services and their Composition (ZEUS 2022)*. Ed. by J. Manner, D. Lübke, S. Haarmann, S. Kolb, N. Herzberg, O. Kopp. Vol. 3113. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 68–75 (cit. on pp. 3, 17, 88–90).

[Voe13] M. Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. In collab. with S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, G. Wachsmuth. Page numbers refer to the free PDF edition. CreateSpace Independent Publishing Platform, 2013. ISBN: 978-1-4812-1858-0. URL: http://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf (visited on Sept. 26, 2026) (cit. on pp. 18, 19, 50, 70).

[VPMR25] D. Voigt, L. Podszun, J. Mangler, S. Rinderle-Ma. “HiProX: Highly Efficient Process Execution on IoT and Edge Devices”. In: *Business Process Management: Responsible BPM Forum, Process Technology Forum, Educators Forum (BPM 2025)*. Vol. 565. Lecture Notes in Business Information Processing. Springer, 2025, pp. 184–199. ISBN: 978-3-032-02936-2. DOI: 10.1007/978-3-032-02936-2_14 (cit. on pp. 23–25, 38).

[VVK09] J. Vanhatalo, H. Völzer, J. Koehler. “The Refined Process Structure Tree”. In: *Data & Knowledge Engineering* 68.9 (2009), pp. 793–818. DOI: 10.1016/j.datak.2009.02.015 (cit. on pp. 15, 63).

[Wie14] R. J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8 (cit. on pp. 7, 85).

[Woh14] C. Wohlin. “Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE ’14)*. New York, NY, USA: ACM, 2014, pp. 1–10. DOI: 10.1145/2601248.2601268 (cit. on p. 7).

[ZS24] W. Zhang, D. Strüber. “Tales from 1002 Repositories: Development and Evolution of Xtext-based DSLs on GitHub”. In: *Proceedings of the 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2024)*. IEEE, 2024, pp. 172–179. DOI: 10.1109/SEAA64295.2024.00034 (cit. on p. 19).