CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Controlled Natural Language Development and Evaluation for Automotive Requirements

A Design Science Study

Master's thesis in Computer Science and Engineering

Nazif Kadiroglu
Fadi Abunaj

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Controlled Natural Language Development and Evaluation for Automotive Requirements

A Design Science Study

Nazif Kadiroglu
Fadi Abunaj

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Controlled Natural Language Development and Evaluation for Automotive Requirements
A Design Science Study
Nazif Kadiroglu and Fadi Abunaj

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover:

Typeset in LaTeX
Gothenburg, Sweden 2026

Controlled Natural Language Development and Evaluation for Automotive Requirements
A Design Science Study
Nazif Kadiroglu and Fadi Abunaj
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Automotive software development depends on requirements that are readable by engineers and precise enough to support later analysis, implementation, and validation. In the Volvo Group context, requirements are maintained in a Requirements Engineering Tool across functional, logical, and software levels. They vary in wording, structure, abstraction level, use of tables, and dependence on surrounding metadata and context. This makes it difficult to restructure requirements consistently into a controlled natural language (CNL) and to determine which requirements can be translated and validated without changing the intended meaning or inventing missing information.

This thesis is a design science research study that develops a classification-driven, layered CNL and analysis pipeline for software-level automotive requirements. The artifact consists of a grounded software-level taxonomy, a stable set of recurring semantic patterns with an associated grammar, and a proposed pipeline that separates the main requirement statement from notes and supporting material before classification, translation, and validation. The resulting artifact also makes the scope boundary explicit by distinguishing supported requirement forms from ambiguous, metadata-dependent, or otherwise unsupported cases. This supports earlier and safer review decisions during requirements engineering. The artifact was developed and refined iteratively through requirement review, practitioner input, and successive CNL design cycles.

The evaluation combined practitioner feedback with review of unseen requirements. Engineers generally judged the CNL clearer and less ambiguous than the original wording, although some table-heavy rewrites were less preferred. On unseen requirements, the artifact covered 166 of 198 requirements, which is 83.8% raw coverage. When the 24 unsupported requirements are excluded, the in-scope coverage was 95.4% (166 of 174). This shows that most requirements within the intended scope can be translated consistently while unsupported cases can be identified early instead of being forced into unsafe translation. The main limitations are that the study is grounded in one industrial setting and that the full end-to-end automated pipeline has not yet been evaluated in operation.

Keywords: controlled natural language, automotive requirements, requirements engineering, design science research, taxonomy, Requirements Engineering Tool.

Acknowledgements

First and foremost, we would like to express our deepest gratitude to our supervisor, Jennifer Horkoff, for her dedication, guidance, and support throughout this thesis. Whenever we had troubles, her advice was always helpful and moved us in the right direction.

Special thanks to our company supervisor, Alex Gholamhosseinpour, for supporting us throughout the work, helping us establish contact with experts in the field, and providing valuable industrial guidance. We are also grateful to Volvo Group for giving us the opportunity to conduct this research and for providing the resources needed for the study.

Finally, we would like to thank all participants in this study. Their willingness to share their time, experience, and knowledge has been invaluable to this thesis. Without their contribution, this work would not have been possible.

Nazif Kadiroglu and Fadi Abunaj, Gothenburg, 2026-06-10

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Modern automotive systems rely on many electronic control units (ECUs), embedded computing units responsible for vehicle functions such as control, communication, sensing, and coordination [1]. Their behavior is specified through requirements and design artifacts at several abstraction levels. In the Volvo Group context, these levels include functional requirements, logical requirements, and software requirements. Functional requirements describe high-level vehicle or feature behavior. Logical requirements refine that behavior into design logic and signal relations. Software requirements specify implementation-oriented behavior for software components and ECUs. This thesis works only at the software-requirement level. Chapter 2 gives a more detailed description of these three levels.

Requirement quality matters because automotive development is commonly organized around a V-model. In this model, requirements and design are refined on the left side until implementation, while verification and validation are carried out on the right side through unit, integration, system, and customer-oriented testing [1]. Figure 1.1 shows this structure. Requirement errors that remain unresolved early can spread into later development and validation, where they are more costly to diagnose and correct. Requirements should be clear and structured early enough to support review, implementation, traceability, and validation, and to reduce later rework that costs time and resources.

The requirements studied in this thesis are maintained in a Requirements Engineering Tool and include software-level requirements for a small set of Volvo-internal ECU datasets. For company privacy, these identifiers are anonymized in the thesis and are referred to as ECU_1, ECU_2, ECU_3, ECU_4, and ECU_5. Across these anonymized ECU datasets, similar requirements may appear in different textual forms, rely on tables, refer to external signal definitions, or include supporting notes and surrounding explanatory text. This variation makes requirements harder to interpret consistently and harder to translate into a structured form without changing the intended meaning or inventing missing information. Related automotive requirements-engineering work reports similar challenges around requirements breakdown, traceability, and verification and validation in complex vehicle-system settings [2].

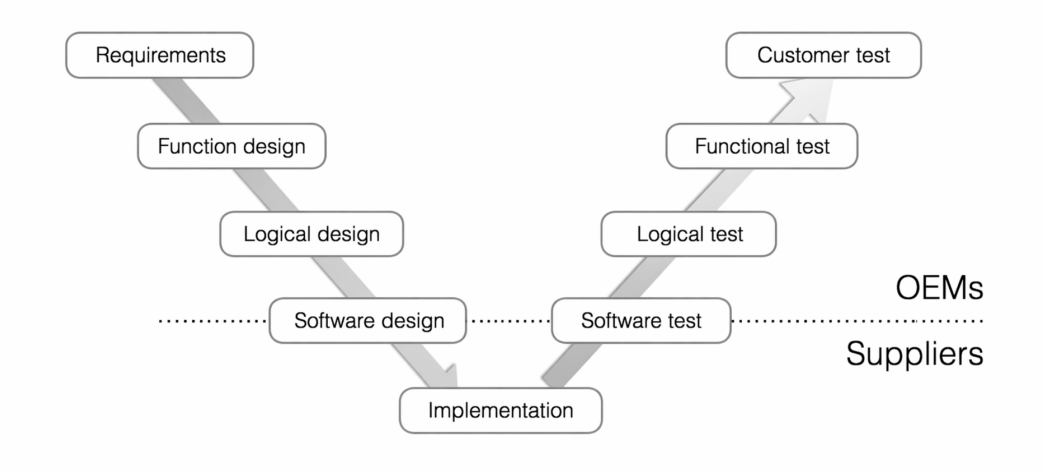


Figure 1.1: V-model structure for automotive development, highlighting how early requirement and design activities on the left side connect to later verification and validation activities on the right side.

Controlled natural languages (CNLs) are relevant because they restrict natural language to improve clarity and machine interpretability while preserving human readability [3]. Existing approaches such as EARS and FRET/FRETISH show that constrained requirement syntax can reduce ambiguity and support later formal analysis [4], [5]. However, applying a CNL in this industrial setting requires more than choosing a template. The supported language must be based on the requirement structures that actually occur in the datasets, and it must make explicit which requirements are supported, which need human review, and which are unsupported. Applying a CNL in this context therefore requires a clear basis from reviewed requirements, and that basis does not yet exist for this material.

1.2 Problem Statement

The problem addressed in this thesis is the lack of a sufficiently clear basis, built from reviewed requirements, for restructuring, parsing, and validating software-level automotive requirements. Existing artifacts may be ambiguous, incomplete, author-dependent, or dependent on metadata and trace links outside the sentence itself. Some artifacts also contain supporting text, such as marked notes or unmarked explanatory context, that should not be mixed with the main requirement statement during translation.

Without a taxonomy of recurring requirement structures, CNL development may become arbitrary. The grammar may support easy examples while missing common industrial forms, or it may overclaim support for requirements that cannot be translated without changing the intended meaning or inventing missing information. The thesis therefore requires empirical classification before later primitive discovery, CNL design, and pipeline design.

1.3 Aim and Research Questions

The aim is to develop a classification-driven CNL basis for Volvo software-level automotive requirements, including a layered grammar basis based on recurring primitive meanings, and to define a pipeline for checking existing requirements against that basis.

The study is guided by three research questions.

- **RQ1 (Problem investigation).** What software-level requirement classes can be derived from reviewed and translated industrial automotive requirements, and what structural properties describe them?
- **RQ2 (Artifact design).** How can the derived classes and recurring primitive meanings be used to build a layered CNL and grammar basis, and how should a pipeline use this basis before translation and validation?
- **RQ3 (Evaluation).** To what extent do the taxonomy, primitive set, layered CNL, and current validation steps support stable and usable CNL design, and what limitations remain?

2

Background and Related Work

2.1 Automotive Development Context

Automotive software is developed in a domain with distributed embedded systems, strong quality constraints, and multiple interacting abstraction levels. Modern vehicles contain many ECUs, each responsible for specific functions and coordinated through software and electrical/electronic architectures [1]. From a software-engineering perspective, this means that requirement quality influences local implementation decisions, later integration, test planning, and cross-component coordination.

Automotive development in this context follows the V-model shown in Figure 1.1. Development activities on the left side refine requirements and design toward implementation. Activities on the right side verify and validate the resulting system through progressively larger test levels [1].

As described in Section 1.1, unresolved requirement issues become more costly later in the V-model. This thesis focuses on software-level requirements, which sit at the bottom of the left branch and form the immediate input to implementation. Defects left in these requirements spread directly into code and are expensive to detect and correct during later verification and validation. This is why the thesis concentrates on structuring and analyzing requirements at this level before implementation begins. Related automotive requirements-engineering research also reports continuing specification and validation challenges in advanced vehicle-system development [2].

In the Volvo context, three requirement levels are relevant. They are functional requirements, logical requirements, and software requirements. This level structure is local to the organization. Functional requirements describe intended vehicle or feature behavior at a high level. Logical requirements refine that behavior into design logic and signal relations. Software requirements, in this thesis, refer to Volvo's software-level category and specify implementation-oriented behavior for software components and ECUs. In simplified terms, one functional requirement may be refined into several logical requirements, which may then be decomposed into several software requirements. This thesis works only at the software-requirement level. Functional requirements in this context are largely free text and are out of scope. In Volvo's internal requirement structure, logical requirements are treated as a more abstract, semantically near-equivalent subset of software requirements. A

CNL expressive enough to capture software requirements is therefore expected to capture logical requirements as well, though the reverse does not hold, since logical requirements may omit implementation-specific detail. The artifact in this thesis was still built and validated on software-level material only.

2.2 Requirements Engineering Tool, ECU Requirements, and Metadata Structures

This thesis studies requirements managed in a Requirements Engineering Tool for anonymized ECU datasets referred to as ECU_1, ECU_2, ECU_3, ECU_4, and ECU_5. The relevant data includes software-level requirements together with surrounding design information such as signal references, tables, identifiers, and linked artifacts. In practice, these requirements are not uniform. Some are relatively explicit and directly behavior-oriented, while others are structurally incomplete, algorithmic, or dependent on interpretation of external metadata. This kind of variation is one reason the thesis treats metadata and surrounding context as part of the requirement-analysis problem, not as side information.

This metadata dependence is important because automotive software development increasingly relies on structured models and standardized architectural concepts. Staron emphasizes the importance of AUTOSAR (Automotive Open System Architecture) and related modeling approaches for describing automotive software structures and interfaces [1]. In this setting, the industrial environment uses metadata structures and signal definitions that are compatible with AUTOSAR-like modeling principles. This makes deterministic validation feasible for some requirement classes, because references to signals, values, units, and interfaces can be checked against structured external information instead of being inferred only from free text.

2.3 Controlled Natural Language and Grammar Basis

Controlled natural languages are restricted subsets of natural language designed to reduce ambiguity and improve machine interpretability while remaining readable for people [3]. In requirements engineering, this is attractive because natural-language requirements are often the shared medium between engineers, reviewers, and domain experts, yet natural language also allows vagueness, omitted assumptions, and inconsistent structure.

In this thesis, CNL limits acceptable requirement formulations so that important information becomes explicit and can be handled more reliably. The practical value of such a language depends on readability, on whether the supported structures align with the requirement classes that occur in industrial datasets, and on whether practitioners find the resulting specification usable in their work. A formally strong CNL has limited industrial value if practitioners do not adopt it in practice, which makes practical usability and later practitioner feedback important design concerns.

2.3.1 Structured Requirement Approaches

CNLs have been studied as a middle ground between unrestricted natural language and fully formal notation. Darif et al. review 133 CNL approaches for requirements published between 2000 and 2024 and group them into standalone templates, requirement patterns, elementary templates, and linguistic-rule approaches [6]. Their review reports recurring limits in tool support, domain vocabulary coverage, industrial validation, and later adoption. This gives useful context for this thesis, where domain vocabulary, metadata dependence, and industrial validation are central concerns.

Existing structured approaches show that restricted syntax can improve requirement quality. EARS offers lightweight templates for recurring requirement patterns such as ubiquitous, event-driven, and unwanted behavior requirements [4]. FRET and FRETISH similarly provide structured ways to express requirements so they can be analyzed more formally [5]. Earlier work has also used CNL as an intermediate layer for reducing the semantic gap between requirements and later test artifacts, arguing that a more structured form can improve consistency between what is specified and what is later validated [7]. Requirements-formalisation research makes a similar point from a non-LLM perspective. Structured pattern languages and semantic analyses can reduce ambiguity and support earlier validation, and full validity or verifiability in context still requires additional work [8]. Taken together, these approaches show that reducing linguistic freedom can make requirements easier to analyze and verify. They also mainly target behavioral requirement forms expressed as recurring condition-response patterns [4], [5].

Closest to this thesis methodologically, Veizaga et al. use a grounded qualitative method to build a CNL, Rimay, directly from industrial requirement material in the financial domain [9]. They use saturation to decide when the supported language is stable and validate coverage on unseen specifications. This thesis follows the same empirical approach, but in a different setting with software-level automotive requirements and heavy metadata and table dependence.

In the automotive domain specifically, Bertram et al. translate informal requirements into Structured English using pattern- and scope-based templates with LLM-assisted few-shot translation, and then check consistency with timed computation tree logic (TCTL) formulas and an SMT solver (satisfiability modulo theories) [10]. Requirements that cannot be expressed in Structured English are deliberately left out, with about 75% of their filtered set remaining expressible. This is the closest automotive CNL setting to this thesis and provides a useful comparison point for explicit language boundaries.

Broader work in automated language processing for requirements also reports that these methods are tested less often in real industrial case studies than in controlled research settings [11]. This matters here because the thesis depends on a real industrial dataset and on validation against unseen requirement material from that setting.

Taken together, prior work shows that constrained requirement language can reduce ambiguity and support later analysis, and that a CNL can be derived systematically

from industrial material. What remains less developed is a CNL for software-level automotive requirements whose meaning often depends on surrounding metadata, signal references, and tables, and whose supported scope is made explicit when safe translation is not possible.

2.3.2 Taxonomy as a Basis for CNL Design

Establishing which classes exist requires a taxonomy that distinguishes requirement types. Behavioral and non-behavioral forms therefore need to be defined before the taxonomy rationale can be stated.

In this thesis, behavioral requirement forms describe observable system behavior, usually as input-condition-output or condition-response logic. Non-behavioral forms describe structures, definitions, storage rules, policies, or internal procedures that are not directly expressed as such behavior. Classification is based on requirement meaning and surface wording. A requirement may look like a simple assignment but still fall outside the current CNL target if it depends on storage, timers, workflow state, history, or missing information. Algorithmic requirements are treated as non-behavioral in this thesis because they describe internal procedure or computation steps. This is consistent with formalisation work showing that different requirement types may need different representation mechanisms [8].

A useful CNL depends on empirical classification of requirements. In this project, the CNL is grounded in a taxonomy built from reviewed requirements. The classification identifies recurring software-level requirement families, reveals where requirement structures saturate, and distinguishes behaviorally translatable cases from non-behavioral or ambiguous artifacts.

Taxonomy alone is not sufficient, however, because one source requirement may still contain several semantic units. A broad class can identify the overall structure of a requirement while still hiding fallback, state, storage, timing, or table-driven behavior inside the same artifact. For that reason, empirical classification functions as the starting point for CNL design, while later atomic decomposition and primitive discovery are needed to identify the semantic units that a grammar must actually support.

2.3.3 Grammar-Based Parsing and Intermediate Representation

Formal grammar and parsing can make a CNL usable for automated checks. Lark is a Python parsing toolkit that supports the definition of grammars and the automatic generation of parsers for structured languages [12]. In a grammar-based approach, a CNL statement that conforms to the grammar can be parsed into a tree and then transformed into a structured abstract syntax tree (AST) representation.

This intermediate representation is central for two reasons. First, it makes the translated requirement explicit enough for later checks on signals, operators, values, units, and structural elements. Second, it separates the human-readable CNL surface

form from the representation used for deterministic validation and later analysis. This separation also matters because one grammar basis does not imply one execution model. Direct behavioral logic, workflow/stateful logic, and storage/lifecycle logic can share one controlled surface language while still requiring different semantic interpretation after parsing.

2.4 Requirement Quality and Validation Considerations

Requirement quality standards provide an important background for any attempt to restructure natural-language requirements into a more controlled form. ISO/IEC/IEEE 29148:2018 emphasizes that individual requirements should be unambiguous, complete, singular, and verifiable, and that sets of requirements should be consistent [13]. These characteristics matter because ambiguity forces interpretation, incompleteness leaves essential information unstated, compound statements weaken traceability, unverifiable wording remains weak even if it is rewritten into a cleaner form, and inconsistency can make later analysis misleading. The same standard also warns against vague pronouns, ambiguous logical phrasing, open-ended non-verifiable expressions, and incomplete references [13].

For this reason, a structured requirement language should not be treated as a cure in itself. Survey work on requirements formalisation and validation shows that pattern-based specification languages, semantic analyses, and structured requirement representations can reduce ambiguity, underspecification, and inconsistency, while full validity or verifiability in context still requires additional confirmation [8]. In other words, structure helps, but it does not by itself prove that the intended meaning has been preserved.

This distinction is especially important in industrial settings where requirements depend on surrounding metadata and linked artifacts. A requirement may look structurally clear while still relying on external signal definitions, tables, units, or trace links for correct interpretation. In such settings, parsing and structural checking remain useful because they make the requirement form explicit and enable deterministic checks on references and well-formedness. Even so, structural correctness is not the same as semantic correctness, and later validation still needs to address meaning, not only syntax.

2.4.1 Automated Support in Related Work

Recent work also highlights limits of automated support for formal or semi-formal translation tasks. Du et al. report that long inputs can degrade LLM task performance even when the relevant information is retrievable, which suggests that long and overloaded artifacts are harder to process reliably [14]. Danso et al. show a related limitation in translation from natural language to linear temporal logic (LTL). Models perform better on syntactic aspects than on semantic ones, so syntactic validity is not reliable evidence of meaning preservation [15]. Broader benchmark

literature also warns that model performance is time-limited, prompt-sensitive, and uneven across evaluations [16], [17], [18], [19].

Other work explores multi-agent debate as one way to improve or assess requirement-related reasoning tasks. Oriol et al. report that debate can improve requirement classification results compared with a single-agent baseline, while also discussing the cost of deeper debate settings [20]. Chun et al. report similar tradeoffs in adjacent software-engineering tasks, where structured debate can improve outcomes but also adds interaction overhead [21]. Liu et al. describe debate-based agent evaluation more generally as a reusable LLM design pattern with both inspectability benefits and computation cost [22]. Taken together, this literature suggests that automated support may be useful, but reliability and meaning preservation remain open concerns.

3

Methodology

3.1 Design Science Research

The thesis follows design science research (DSR) because the objective is to construct and evaluate an artifact that addresses a practical software-engineering problem [23]. In this thesis, the artifact consists of a requirement taxonomy, primitive labels, a CNL and grammar basis with layer-specific execution models, and a pipeline design that together support safer transformation of varied automotive requirements into a structured form.

Knauss describes constructive master's thesis work in industry as a systematic and iterative method that identifies a problem, develops a design artifact, and evaluates the artifact in context [24]. The study follows this logic through four completed design cycles. Each cycle contains problem investigation, artifact design, and evaluation, but the emphasis differs across cycles. Figure 3.1 summarizes the four design cycles and their main study activities. The Cycle IV validation tracks are described in Section 3.5.

3.2 Industrial Context

The study is conducted in collaboration with Volvo Group. The requirement artifacts are managed in a Requirements Engineering Tool and include software-level requirements associated with several ECUs. In this thesis, ECU means electronic control unit. For company privacy, Volvo-internal ECU and system identifiers are anonymized and are referred to as ECU_1, ECU_2, ECU_3, ECU_4, and ECU_5. In this context, requirements are linked to design information such as signal definitions, tables, identifiers, and related artifacts.

The thesis uses Volvo's local distinction between functional, logical, and software requirements, as described in Section 1.1. In this thesis, software requirements means Volvo's software-level category. The artifact was built and validated on software-level material only. Functional requirements are outside the scope of the study. Logical requirements were discussed during scoping because they are a semantic subset of software requirements, but they were not part of the analyzed material. Because these categories are context-specific, the classification and CNL design in this thesis are evaluated against the software-level Volvo material.

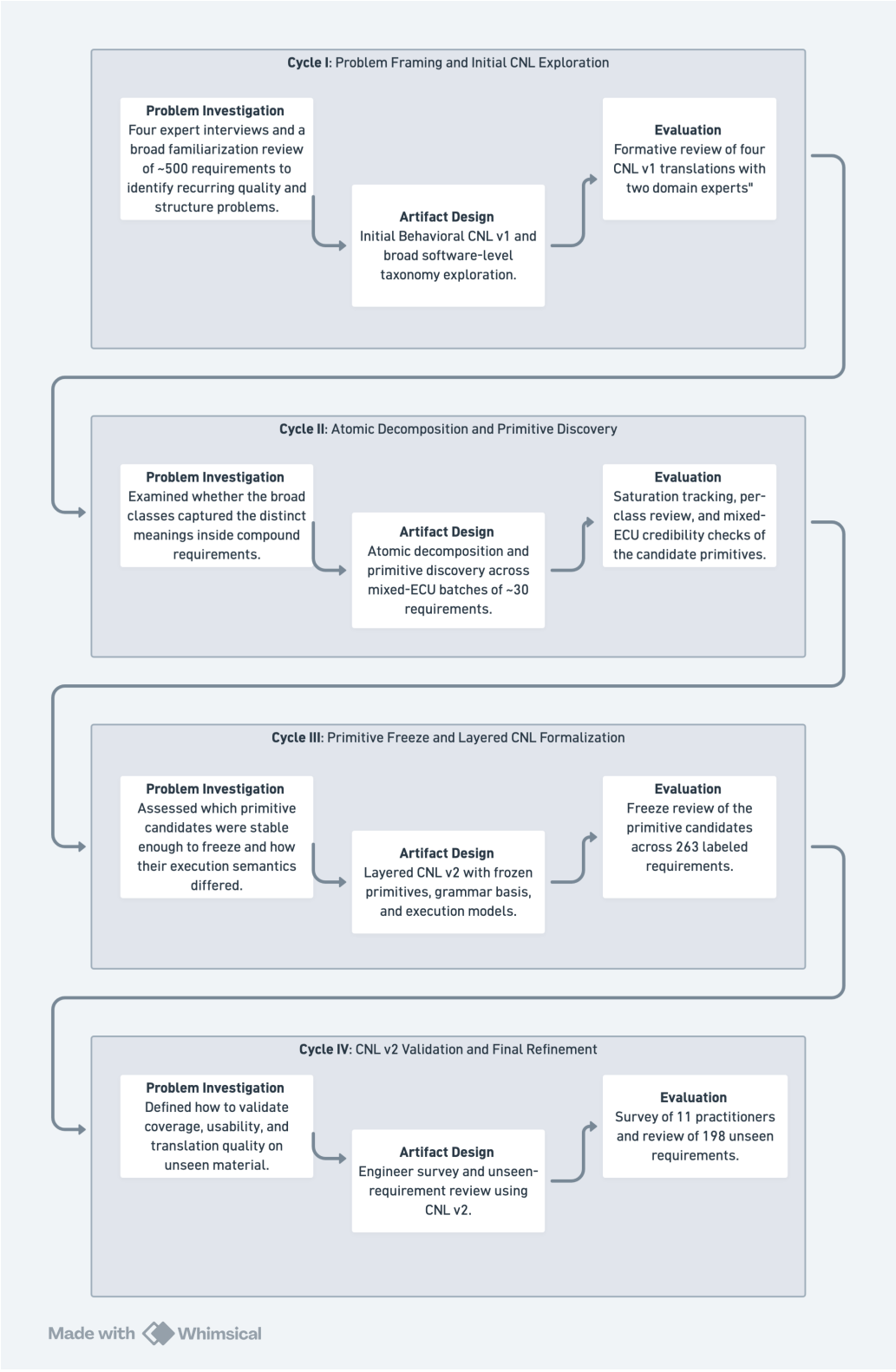


Figure 3.1: Overview of the four DSR cycles in the thesis, showing the problem investigation, artifact design, and evaluation focus in each cycle.

3.3 Data Collection

The empirical material combines interviews and requirement datasets. The interviews provided problem context and design input from practitioners who work with requirements in the organization. The requirement datasets provided the material for classification, translation, and validation work.

3.3.1 Interviews

Four semi-structured interviews were conducted with practitioners who work with requirements in the organization. Semi-structured interviews were selected because they allow consistent coverage of planned topics while leaving room for clarification and probing [25]. The interview guide is included in Appendix A.1. Participants were selected through purposive sampling. The goal was to include people with different requirement-related perspectives. Therefore, participants were selected because they write, review, test, or use requirements in their daily work. All interviews were formal and recorded.

At an early stage of Cycle I, three informal scoping discussions were held with domain experts representing software, logical, and functional design perspectives. These discussions were used to understand how requirements are expressed at different levels and to assess where a CNL pipeline was most feasible. They provided scoping input before the formal interview sample reported in Table 3.1. After CNL v1 was drafted, the first of three CNL iterations described in Section 4.1.3, selected early translations from a recurring requirement type were also discussed with domain experts for formative feedback before the later primitive-discovery and layered CNL work.

The interviews were conducted by the two thesis authors. Both authors approached the interviews from a software-engineering and requirements-engineering perspective. Since the authors were not regular members of the interviewed teams, some Volvo-specific terminology and practices had to be clarified during the interviews.

The interview topics covered requirement-writing practices, ambiguity, missing information, traceability, metadata dependence, table representations, supporting notes and context, and feasibility constraints for structured requirement languages.

3.3.2 Requirement Datasets

ECU_1 served as the primary exploratory dataset because it contained enough software requirements for class discovery and CNL translation work. ECU_2, ECU_3, ECU_4, and ECU_5 were used as additional cross-ECU datasets to test the taxonomy against varied wording styles.

ID	Date	Role (anonymized)	Experience	Relevant background
I1	2026-02-25	System design / functional design role	2 years	Reviews requirements, works with functional and system-level requirements, and has written software requirements.
I2	2026-02-26	System architecture role	4 years	Writes architecture-level requirements and works with how these are later implemented across functional, logical, and software levels.
I3	2026-03-28	Senior requirement-related and software development role	11+ years	Has experience across several development areas and has previously developed a requirement language and compiler for logical validation.
I4	2026-03-28	System architecture role	7 years	Works with logical and software design, requirement testing, and platform architecture. Mainly tests and uses requirements.

Table 3.1: Interview participants included in the qualitative analysis.

3.4 Data Analysis

3.4.1 Qualitative Interview Analysis

The interviews were analyzed using inductive thematic analysis based on Braun and Clarke's guidance [26], [27]. The work included familiarization with transcripts, coding of meaning units, grouping of related codes into themes, and interpretation of how the themes shaped the artifact.

Coding was applied to meaning units instead of full pages, and multiple codes could be assigned to the same excerpt when needed. Codes were initially kept descriptive and close to the data, and overlapping codes were then consolidated into a smaller codebook. All four interviews were coded and reviewed against the evolving codebook. No new codes were created after the third interview. The fourth interview reinforced existing codes without introducing new themes. Appendix A.2 contains the detailed codebook. The main body reports the themes and how they shaped the artifact in Chapter 4.

3.4.2 Software-Level Classification Analysis

The software-level classification and translation work was conducted manually by both thesis authors. In this thesis, *reviewed* means that the authors read the original requirement and its surrounding material, *classified* means that the authors assigned the requirement to one of the derived requirement classes, *translated* means that the authors manually rewrote the requirement into the proposed CNL form, *primitive-labeled* means that the authors assigned one or more candidate or frozen primitive labels to the requirement or its atomic statements, *frozen* means that a primitive was accepted into the final supported set, and *validated* means that the requirement was later used in the survey-based or unseen-requirement evaluation tracks. Both authors participated in classification, decomposition, and primitive-label review. Disagreements were discussed until a shared interpretation was reached. When agreement could not be reached without risking a meaning change, the requirement remained in the ambiguous, partially translatable, or unsupported set. The goal was to derive a stable taxonomy that could guide CNL and grammar design. For Cycle I, this class-discovery work followed Mayring's qualitative content analysis as a systematic, rule-guided text analysis with inductive category development from concrete material [28].

The analysis and artifact-development work had seven phases.

1. Manual review, inductive broad-class discovery, and CNL translation of ECU_1 software requirements by both authors.
2. Decomposition of compound requirements into atomic semantic statements.
3. Class-by-class primitive discovery in batches until additional batches no longer yielded new semantic patterns.
4. Consolidation of candidate primitive labels.

5. Construction of a representative translated reference dataset with selected examples for each class.

6. Development of the layered CNL, grammar basis, and execution models.

7. Validation through engineer survey material and unseen requirement review.

ECU_1 was the main dataset for class discovery and early CNL translation. Translation supported both class discovery and grammar development. The broad classes were developed inductively from the ECU_1 material, refined during review, and then applied to additional material [28]. Review continued until the class structure stabilized and then continued further to strengthen confidence in the classification work. After the class structure stabilized, classification was used on the remaining ECU_1 material and all software requirements in the additional ECU datasets. Later translation work focused on class coverage checks and validation coverage. Selected translation samples from ECU_2, ECU_3, ECU_4, and ECU_5 were also used to test whether different wording styles introduced new classes. The specific review and sample counts are reported in Section 4.1.

3.4.3 Pre-analysis and Pipeline Design Rationale

The artifact design also included a pre-analysis stage before any CNL translation. During manual review, the authors first isolated the main requirement statement from notes and surrounding supporting text. Only the isolated main statement was treated as the translation target. Notes and surrounding context were preserved separately because they could support interpretation without always belonging inside the enforceable requirement statement.

The pre-analysis stage also recorded whether the artifact was textual, tabular, or mixed and whether it showed translation-relevant flaws. These flaw categories were grounded in ISO/IEC/IEEE 29148:2018 and were used in this thesis as manual review criteria. The recorded flaws included ambiguity, incompleteness or missing references, lack of singularity through compound statements, unverifiable wording, inconsistency, and incomplete signal information when relevant. Flaw detection was therefore still part of the study, but it was carried out manually by the authors during artifact development and validation. The thesis did not evaluate automated flaw detection.

The pipeline design kept structural and semantic checks separate. Parsing and metadata-based checks address structure, references, and well-formedness. Meaning preservation needs a separate later assessment. The designed pipeline also leaves room for future automated support such as focused LLM checks, automated translation, and debate-based semantic assessment. However, those automated stages were not implemented or evaluated in this thesis. In this study, classification, flaw detection, translation-safety judgments, translation, and validation logic were applied manually by the authors, while the final CNL grammar itself was implemented in Lark.

3.4.4 Primitive Discovery, Freeze Review, and Validation

Cycle II and Cycle III extended the class-based analysis with atomic decomposition and primitive discovery. A class groups requirements by structure. A primitive label captures recurring behavior inside or across classes. Many source requirements contained more than one semantic statement, so compound requirements were decomposed into atomic statements before primitive discovery. This moved the study from whole requirements to smaller semantic units. Cycle I focused on inductive category development to identify top-level requirement families [28]. Cycle II then compared decomposed atomic statements with existing primitive candidates to identify recurring semantic patterns inside those families [29]. Hennink et al.'s distinction between code saturation and meaning saturation is useful for explaining this shift [30]. In this thesis, Cycle I focused on category saturation. Additional reviewed requirements no longer introduced new top-level requirement families. Cycle II focused on deeper semantic saturation. Additional decomposed atomic statements no longer introduced new semantic patterns.

Primitive discovery was performed class by class in mixed-ECU batches of about 30 requirements so that the emerging labels would not reflect only one ECU's wording habits. Each decomposed atomic statement was compared with existing primitive candidates by constant comparative logic [29]. If a batch introduced a new semantic pattern, a new candidate was added and the next batch was reviewed. Saturation in Cycle II meant that a new batch inside the same broad class did not introduce a new semantic pattern. One extra confirmation batch was then reviewed after the first zero-new batch to reduce the risk of stopping too early.

This work produced candidate primitive labels that were later consolidated and reviewed. Cycle III then shifted from discovery to freeze review. In this thesis, to freeze a primitive means to treat it as no longer provisional. It is accepted into the final supported primitive set and linked to a defined CNL form and semantic layer. A candidate primitive was frozen into CNL v2 only when repeated examples showed one stable meaning, one consistent CNL form that did not change the intended meaning or invent missing information, and a clear fit inside one execution layer. Candidates that kept changing definition, crossed layer boundaries, or required unsupported interpretation were not frozen as part of the grammar. This freeze review produced the final layered primitive set and CNL basis reported in Section 4.3 and Table 4.2.

As part of Cycle III, a later semantic review assigned primitive labels to reviewed requirements according to their main logical behavior and was used to freeze the stable primitive set. Cycle IV then validated the resulting artifact through two separate tracks. One was a survey answered by 11 practitioners using ten selected original requirement and CNL translation pairs. A translation pair is the original requirement together with its CNL translation. The other was an unseen-requirement review of 198 requirements that had not been used during primitive discovery. The unseen validation set was a non-random set of 198 requirements. It gave rough spread across the derived classes and included structurally difficult or compound cases. The coverage values show how the artifact handled difficult unseen material, not all software requirements. The unseen review reported both raw coverage and

in-scope coverage so that total performance and scope-bounded performance could be interpreted separately.

3.5 Evaluation Strategy

The evaluation covers software-level requirements only, as scoped in Section 3.2.

Evaluation in this thesis has three connected parts. The first part focuses on whether the derived software-level taxonomy covers the reviewed and classified requirements. In Cycle I and Cycle II, the main criteria are whether broad classes and later primitive structures stabilize during additional ECU_1 review, selected cross-ECU translation checks, and later all-ECU classification, and whether reviewed requirements can be placed within the taxonomy or in an explicit incomplete/ambiguous category.

The second part focuses on primitive stability and layer consistency. The main criteria are whether a primitive has consistent meaning across examples, whether translation rules remain reliable, whether different execution semantics are kept separate, and whether unsupported behavior is kept outside the CNL.

The third part focuses on validation of usability and coverage. The engineer survey examines readability, wording, and preference across ten selected translation pairs and provides formative evidence about clarity, ambiguity, and practitioner preference. The unseen-requirement review examines raw and in-scope coverage on 198 requirements that were not used during primitive discovery. The unseen-requirement review was used as the main empirical basis for the coverage claim.

4

Results

Artifact Overview

The final artifact developed in this thesis is a CNL approach for automotive requirement analysis that is built on requirement classification. It has five connected elements.

1. A software-level requirement taxonomy.
2. Primitive labels for recurring requirement behavior.
3. A translated example set drawn from the identified requirement classes.
4. A software-level CNL v3, its grammar, and rules for how each layer should be interpreted.
5. A pipeline design for preparing requirements before translation, parsing, and validation.

These elements are treated as one artifact because each one depends on the previous one. The taxonomy defines the requirement families. Primitive labels identify recurring behavior inside those families. The translated example set shows how those behaviors can be rewritten in a controlled form. CNL v3 and its grammar are built from the frozen primitives and define how each layer should be interpreted. The pipeline then uses that basis to review Requirements Engineering Tool artifacts before translation, parsing, and validation.

Class–Primitive Distinction. Classes describe the structure of a requirement. Primitive labels describe the behavior expressed inside that structure. This distinction matters because two requirements can belong to the same class but still require different CNL treatment.

For example, two requirements may both belong to the same broad class. The first requirement expresses a simple condition-action behavior.

> *"If Seatbelt is unbuckled, Warning shall be set to Active"*

The second requirement also has a conditional structure, but it includes target-local retained-value behavior. The output keeps its previous value when no new valid

value is used. In this example, retention belongs to the same output and does not require inventing a separate stored source.

> *"If Signal is valid, Output shall be set to Signal, otherwise Output shall retain its previous value"*

The class is the same, but the primitive is different. The taxonomy was used to group requirements during analysis, while the CNL itself was defined by the final primitives and the rules attached to each layer. This distinction helped separate stable patterns from cases that were still too unclear or too different to support safely.

4.1 Cycle I Findings

Cycle I was the first design science cycle in the thesis. Its purpose was to understand the industrial problem, choose the first workable requirement level, and derive the broad software-level classes that later structured the artifact. Cycle I did not aim to build the final software-level CNL. Its main outputs were a finalized classification scheme for software-level requirements (see Figure 4.2 and Table 4.1), an early Behavioral CNL v1, and the insight that broader classification had to come before stable grammar design.

4.1.1 Problem Investigation

Cycle I began with four formal interviews, continued with a broad familiarization inspection of about 500 requirement artifacts from this environment, and then moved into a focused ECU_1 class-discovery review of about 330 software requirements. The interviews identified recurring practical problems. The broad inspection checked those issues against a wider requirement set. The focused ECU_1 review then turned the early observations into class-discovery work. The findings are presented by theme because the same issues appeared across Interviews I1–I4 and across the early requirement material.

Uneven standardization and author-dependent writing. Some teams used templates or local conventions, while others wrote requirements more freely. This meant that similar meanings could appear in different textual forms. The variation weakened consistent interpretation, translation, traceability, and validation. It also supported the need for a controlled target language that could make recurring structures more explicit. The artifact therefore defines one CNL form that can be checked even when the original requirements are written in different ways.

Abstraction level and scope choice. Functional requirements were described as broader and more dependent on interpretation. Logical requirements were more abstract than software requirements, but still close to signal behavior and logical relations. Software requirements were often more explicit because they referred to signals, inputs, outputs, and expected behavior. This made software requirements

the safest first target for parser-oriented CNL work in Cycle I, while functional requirements were kept outside the first grammar target. The semantic closeness of logical requirements suggested that a software-level CNL might later reach them as well, but the first artifact scope stayed at the software-requirement level.

Metadata, traceability, and external dependence. Practitioners repeatedly connected validation to signal definitions, linked artifacts, databases, glossaries, abbreviations, and trace links. Surface wording alone was often not enough for reliable interpretation. A requirement could look understandable but still be difficult to validate if signal references, units, or links were missing or incomplete. This insight later shaped the pipeline thinking, where translation could not depend on sentence wording alone, which is why the artifact preserves traceability, checks metadata, and flags missing information explicitly.

Notes, context, and supporting text. Participants explained that notes and surrounding text could help some readers understand the requirement, especially when the audience was broader than the original author. At the same time, they also said that such material should not always be merged into the enforceable requirement statement. In some cases, notes were redundant, misplaced, or closer to explanation than to software-level behavior. This later motivated the separation between the normative requirement statement and optional supporting context and led to separate handling of supporting text.

Tables and mixed artifacts. Tables were sometimes useful for repeated mappings or structured signal relations, but they were not always semantically simple. Some tables expressed straightforward mappings, while others expressed priority rules, state-based behavior, or ambiguous legacy structures. This meant that tables could not be translated as plain text without risking a meaning change unless they were first classified by the logic they expressed, which shaped the artifact boundary by requiring table forms to be classified before any translation work.

Limits of formalization. Formalization was seen as useful, and one senior participant described prior experience with a requirement language and compiler that could check logical correctness. At the same time, practitioners also pointed out that some requirements were too exceptional, underspecified, or complex to force into one standard form without changing the intended meaning. This supported the later decision that the CNL would need explicit exception handling and human review instead of assuming universal coverage. This was a smaller theme, but it reinforced the need for explicit exception handling and human review for unclear or unsupported cases.

Across Interviews I1–I4, the same main issues recurred and strengthened the evidence that uneven writing, missing clarity, and extra information inside requirements were practical problems across the organization.

The full interview codebook is provided in Appendix A.2.

These interview themes were then checked against the early requirement material and later refined through the focused ECU_1 review. The early broad familiarization inspection supported the interview findings. About 500 requirements were inspected during this phase. Repeated assignment-style patterns were noticed early, especially direct assignments and condition-action rules. Many requirements were also signal-heavy and referred to inputs, outputs, values, states, and external definitions. At the same time, many artifacts were compound, mixed text and table structures, or depended on context outside the requirement sentence. The nine broad classes later used in the thesis began to emerge during this familiarization phase as broad organizing groups.

The focused class-discovery review then used this dataset as the main review set. About 330 ECU_1 software-level requirements were manually reviewed, classified, and translated into the early CNL during this phase. The purpose was to discover the broad top-level requirement classes and test whether recurring requirement structures could be rewritten into a controlled form. This step followed Mayring's inductive category-development logic. Broad classes were derived from concrete requirement material, revised during review, and later applied to further material [28]. No new top-level classes appeared after about 200 reviewed requirements, which showed class-level saturation, but the review continued to around 330 requirements to strengthen confidence in the result.

Figure 4.1 shows the observed saturation profile during the Cycle I class-discovery review.

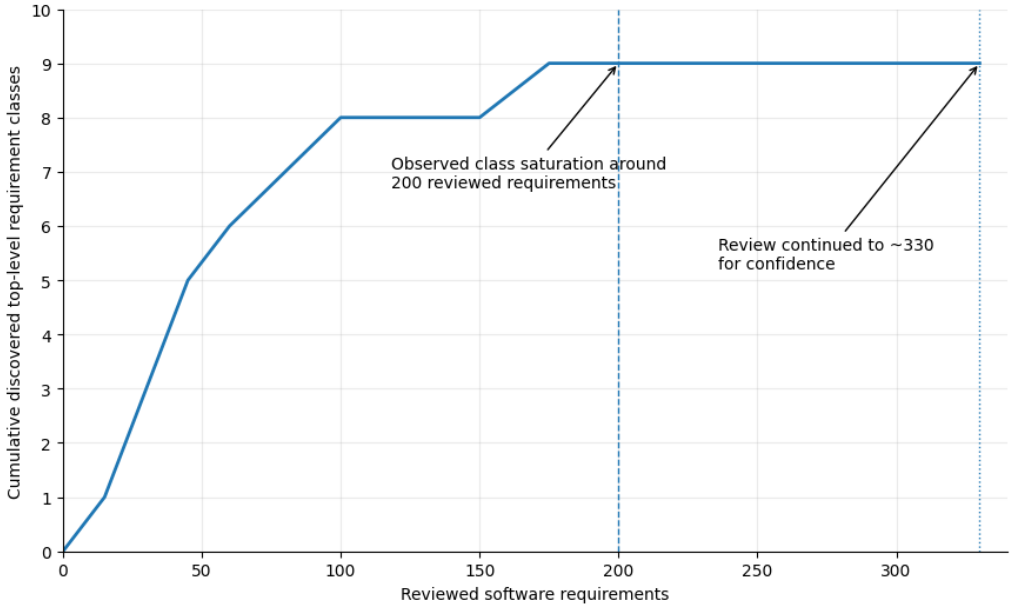


Figure 4.1: Observed saturation profile during iterative review of the ECU_1 class-discovery material in Cycle I.

After the ECU_1 class structure had stabilized, selected cross-ECU translation checks were carried out on ECU_2, ECU_3, ECU_4, and ECU_5. These samples consisted of 50 requirements from ECU_2, 40 from ECU_3, 10 from ECU_4, and

20 from ECU_5. The selections were made deliberately to cover varied structures and to avoid sampling only frequent patterns such as conditional assignment. No new broad classes appeared in these additional cross-ECU samples.

4.1.2 Software-Level Requirement Taxonomy

The software-level taxonomy contains nine families.

1. Unconditional Assignment
2. Conditional Assignment
3. Priority Table Evaluation
4. State-Machine / Timed Behavior
5. Settings / Storage Management
6. Algorithmic Computation
7. Service / Control Policy
8. Enum / Type Definition
9. Incomplete / Ambiguous Artifact

These nine families are high-level requirement families used during classification. They describe what kind of requirement artifact is being reviewed before deciding whether it fits the Behavioral layer, another supported layer, or an unsupported category.

Following the distinction introduced in Chapter 2, the taxonomy separates the reviewed requirements into three groups. The first group contains families that can be handled directly by the Behavioral and Workflow / Stateful layers. The second group contains families that need different handling. Settings / Storage Management is handled through the Storage / Lifecycle layer, while Algorithmic Computation, Service / Control Policy, and Enum / Type Definition remain outside the current grammar target. Incomplete or ambiguous artifacts lack enough information for translation without changing the intended meaning and are therefore flagged.

The most common supported pattern is condition-action logic, where current conditions are checked and one or more outputs are assigned.

Figure 4.2 visualizes these taxonomy groups and their scope boundaries.

After this final class system had been derived, all software requirements in the ECU_1, ECU_2, ECU_3, ECU_4, and ECU_5 datasets were classified against the taxonomy. This later classification stage was used to apply the final category system across all software requirements in those datasets and to check that no further broad classes were needed.

Figure 4.3 summarizes how Cycle I combined class-discovery review, selected translation samples, and later full-dataset classification across the five ECU datasets.

Family	Description
Unconditional Assignment	A requirement that assigns a value directly, without an explicit condition.
Conditional Assignment	A requirement that assigns a value only when one or more conditions hold, sometimes with fallback behavior.
Priority Table Evaluation	A requirement where several rules, rows, or input combinations are evaluated in a priority order.
State-Machine / Timed Behavior	A requirement that depends on states, transitions, timers, delays, counters, or time-based behavior.
Settings / Storage Management	A requirement that involves stored values, defaults, saving, restoring, initialization, or persistent storage.
Algorithmic Computation	A requirement where the output is calculated through a formula, transformation, filtering, accumulation, or conversion.
Service / Control Policy	A requirement that describes service interaction, request-response behavior, diagnostic policy, publishing policy, or interface behavior.
Enum / Type Definition	A requirement that defines possible values, types, states, constants, or allowed enumerations.
Incomplete / Ambiguous Artifact	A requirement artifact that is unclear, incomplete, contradictory, or missing information needed for translation without changing the intended meaning. Translation would require guessing, inventing missing information, or changing the meaning.

Table 4.1: Brief descriptions of the software-level requirement families.

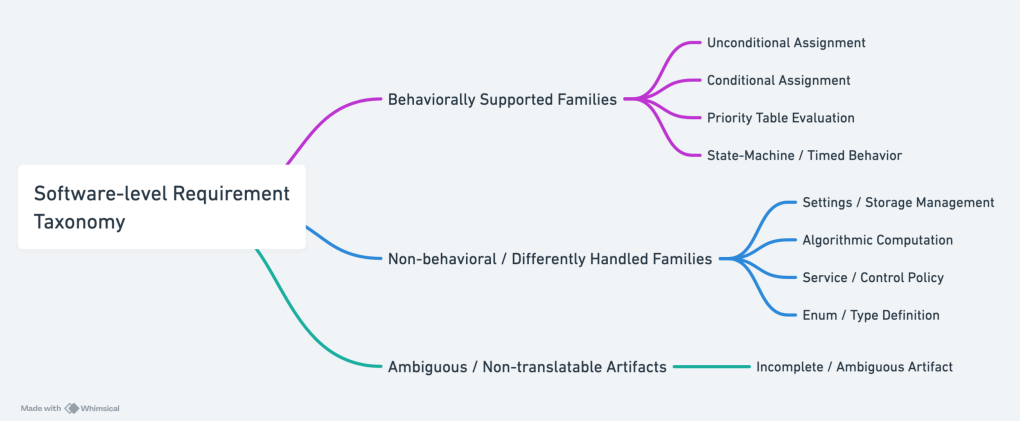


Figure 4.2: Derived software-level taxonomy, distinguishing behaviorally supported families, differently handled families, and incomplete or ambiguous artifacts.

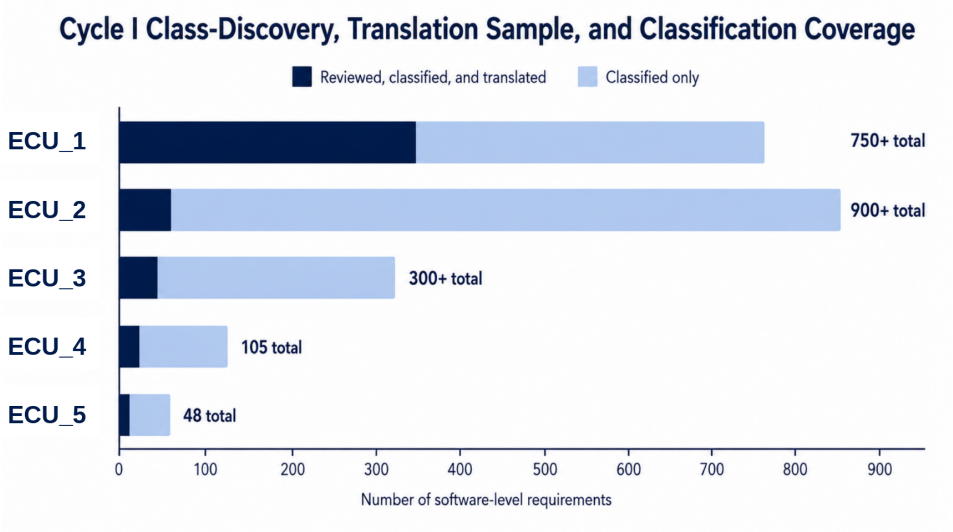


Figure 4.3: Cycle I class-discovery, translation-sample, and classification coverage across the ECU datasets. Dark blue shows requirements that were manually reviewed, classified, and translated during class discovery or selected cross-ECU translation checks. Light blue shows additional requirements that were classified against the final taxonomy after the class structure had stabilized.

4.1.3 Artifact Development

Cycle I produced an early Behavioral CNL v1 as a feasibility test. The purpose was to test whether recurring assignment-style software requirements could be rewritten into a more consistent, readable, and machine-checkable form without changing their meaning.

The scope of CNL v1 was intentionally narrow. It covered unconditional assignment, conditional assignment, simple condition-action assignment, and explicit fallback only when the source requirement stated the fallback directly. It did not cover the other main requirement families, such as workflow or stateful behavior, storage or lifecycle behavior, and algorithmic computation. It also did not cover compound or ambiguous requirements.

The first form covered direct assignment.

```
<Target> shall be set to <Value>.
```

The second form covered condition-action logic.

```
When <Condition>, then <Target> shall be set to <Value>.
```

The third form covered explicit fallback when the source requirement already stated the fallback case.

```
When <Condition>, then <Target> shall be set to <Value1>.

Otherwise, <Target> shall be set to <Value2>.
```

Figures 4.4 to 4.7 show four representative example pairs from this early feedback

round. Each pair contains the original requirement followed by its CNL v1 translation. Although these rewrites are simple, they still show the value of a controlled form. One uniform syntax makes the requirement logic easier to compare, review, and interpret across source requirements with different wording.

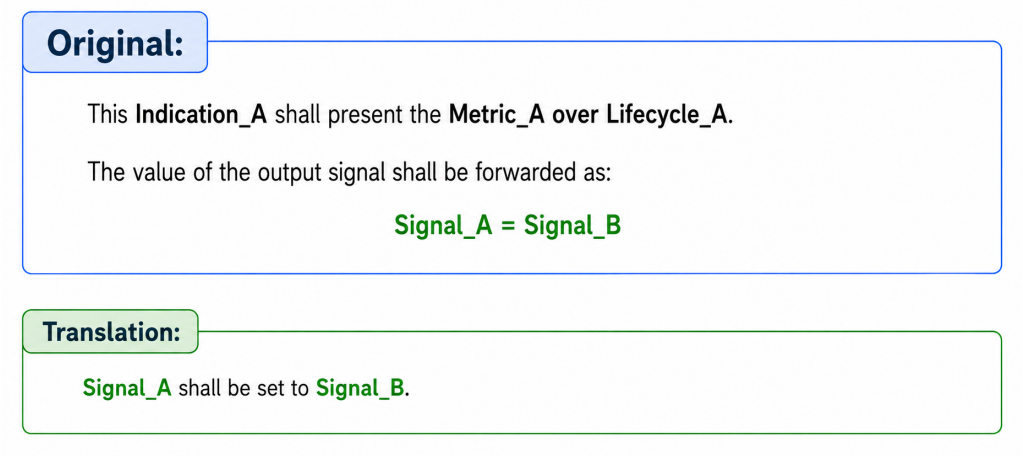


Figure 4.4: Original requirement and CNL v1 translation for an unconditional forwarding rule where `Signal_A` is set to `Signal_B`.

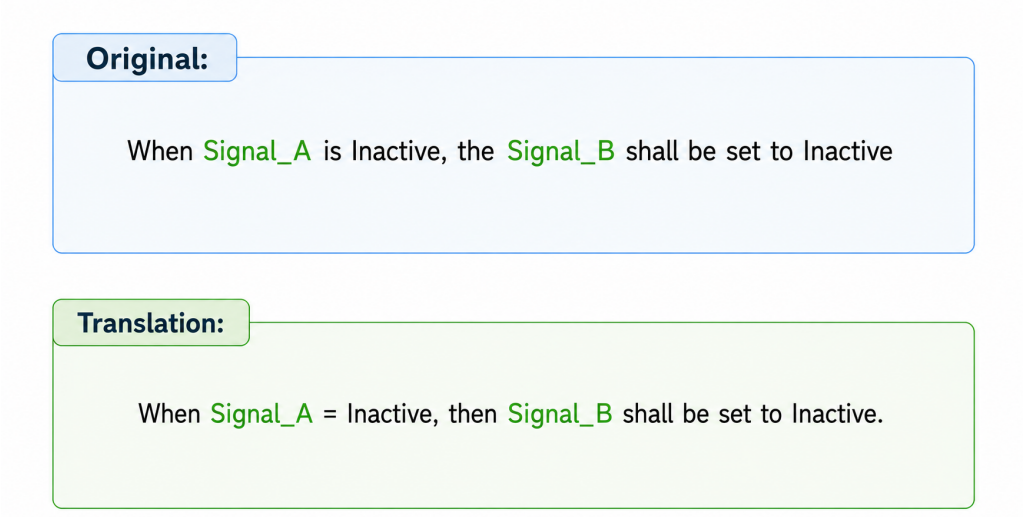


Figure 4.5: Original requirement and CNL v1 translation for a conditional assignment where `Signal_A` is set to `Inactive` when `Signal_B` is `Inactive`.

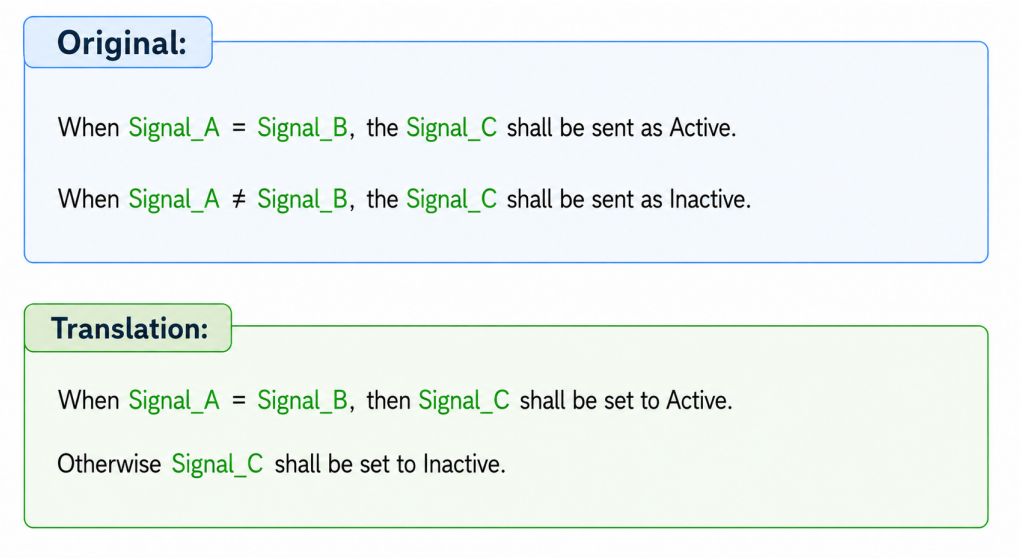


Figure 4.6: Original requirement and CNL v1 translation for a conditional assignment with an explicit fallback, where `Signal_A` is set to `Active` in the named operating mode and otherwise set to `Inactive`.

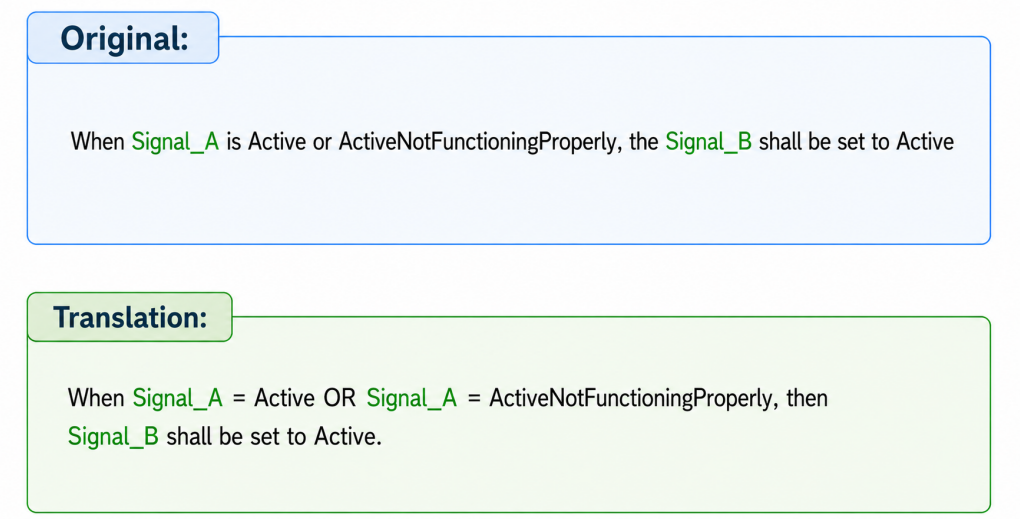


Figure 4.7: Original requirement and CNL v1 translation.

4.1.4 Evaluation and Main Lesson

Four original requirements and their CNL v1 translations were discussed with two domain experts who had participated in the Cycle I interview phase. The selected examples focused on the narrow assignment-style cases supported by CNL v1.

The feedback covered readability, meaning preservation, whether the syntax was too narrow, whether extra context was needed, whether supporting text should stay separate from the CNL sentence, and whether broader taxonomy work was needed before grammar expansion. For the selected examples, both experts judged all four

translations readable and accurate. They did not identify a need for extra notes or context for these specific cases, and they said that supporting text could remain separate instead of being merged into the CNL sentence. Both experts preferred the CNL version over the original requirement in all four cases.

The main Cycle I lesson was to treat assignment-style cases as a first step. The next step was to support more requirement types and more complex cases, but this first required a broader classification of requirement families, followed by splitting compound requirements into smaller meaning units and then identifying recurring primitive meanings. Cycle I therefore established assignment-style CNL as a useful first step, but also showed that compound requirements and mixed semantics had to be classified and decomposed before stable grammar design. Cycle II moved from broad classes to atomic decomposition and to identifying recurring primitive meanings.

4.2 Cycle II Findings

4.2.1 Problem Investigation

Cycle II identified recurring primitive meanings inside the broad classes from Cycle I. An atomic statement is one small meaning unit taken from a requirement. A primitive label is the name given to a recurring kind of atomic statement. Cycle II built on the inductive category work from Cycle I [28], but the analysis depth changed. Cycle I asked when reviewed requirements stopped introducing new top-level classes. Cycle II asked when decomposed atomic statements stopped introducing new semantic patterns inside those classes. As noted in Section 3.4.3, Hennink et al.'s distinction between code saturation and meaning saturation is useful for explaining this shift [30].

The main issue in Cycle II was that one requirement could contain several different meanings. A broad class such as conditional assignment could still hide several recurring structures. Some requirements combined normal assignment, fallback behavior, timing, storage behavior, table-driven logic, and explanatory text in one artifact. For example, one requirement could contain a normal conditional assignment, an invalid-signal fallback, and a table-based fallback in the same requirement. An example of this is shown later in Figure 4.10. This made it necessary to move from requirement-level grouping to atomic semantic analysis.

4.2.2 Artifact Development

Cycle II produced a method for splitting compound requirements into atomic statements, a method for finding primitive labels within each broad class, a batch review that mixed requirements from several ECUs, an early set of candidate primitive labels, and the results later used in Cycle III. The goal was to discover recurring semantic structures without forcing the requirements into final templates too early.

The overall Cycle II procedure is summarized in Figure 4.8. The figure makes explicit that Cycle II linked the broad class taxonomy from Cycle I to the later Cycle III review that decided which primitives could be kept. The unit of analysis changed from whole requirements to atomic semantic statements, and the stopping rule changed from saturation of broad classes to saturation of new semantic patterns.

The 36 consolidated primitive candidates were not one set per class. A broad class could contain several different primitives, and some primitives could recur across more than one class.

An atomic statement was defined as one requirement-level semantic unit that could potentially receive one primitive label and later be represented by one CNL primitive. A primitive label belongs to the discovery stage. A CNL primitive is a label that was later frozen into the language. This definition became necessary because one requirement could contain more than one meaning. Decomposition was needed both for discovery and for later translatability. Without decomposition, a single compound requirement could appear to belong to one broad class even though it

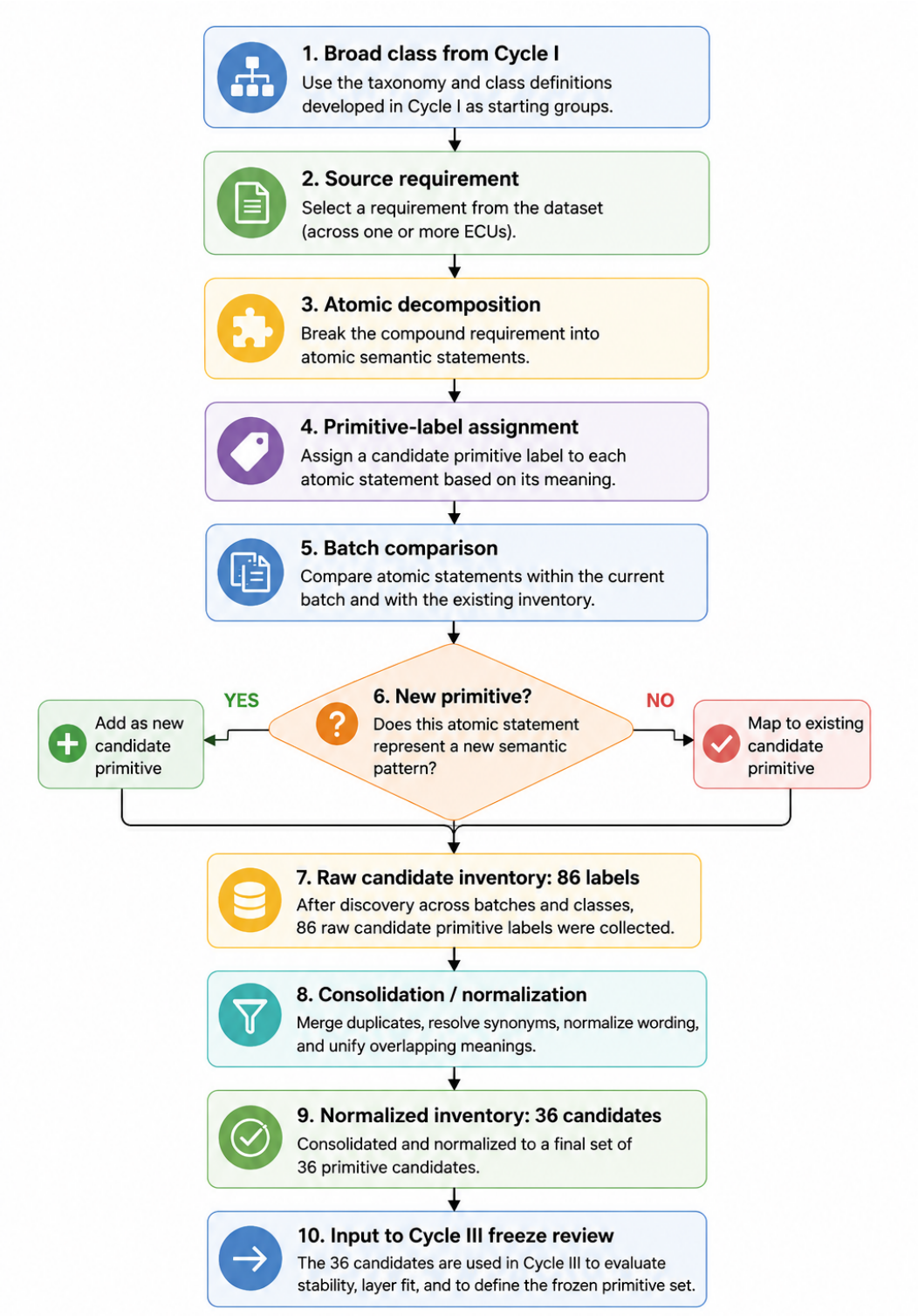
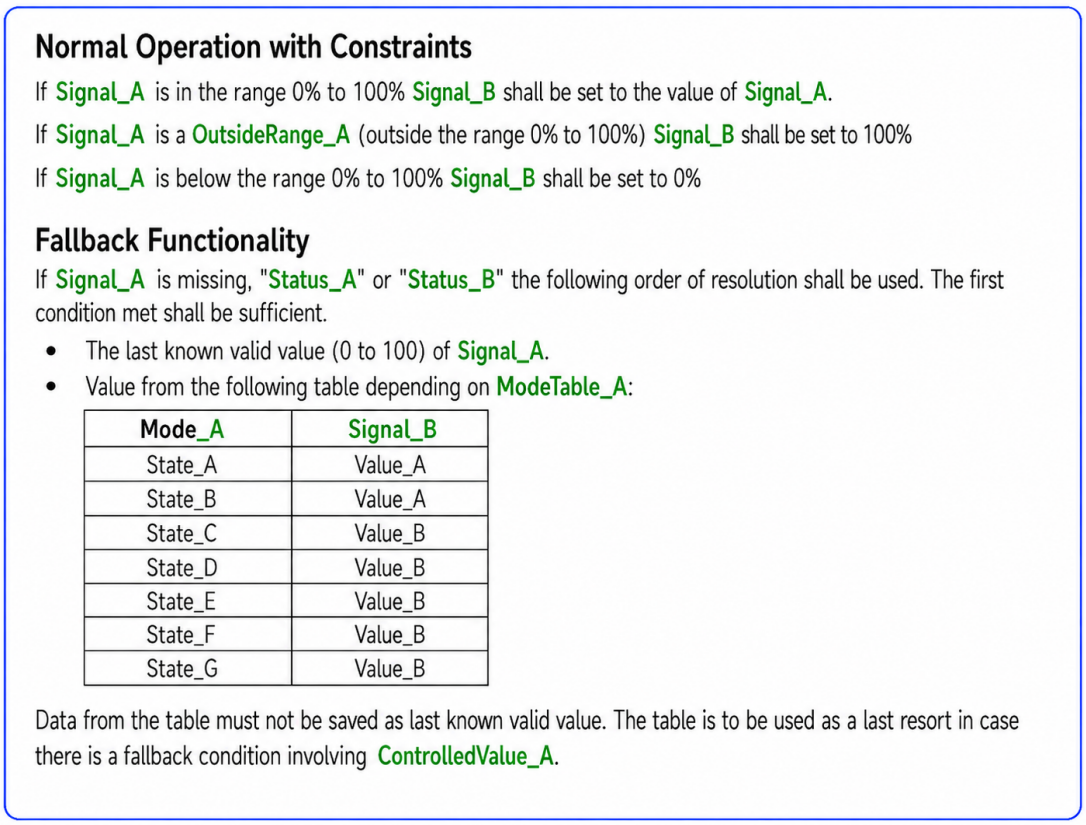


Figure 4.8: Cycle II primitive-discovery workflow. Broad requirement classes from Cycle I were used as starting groups. Requirements were decomposed into atomic semantic statements, assigned candidate primitive labels, compared across mixed-ECU batches, and then consolidated from 86 raw labels to 36 consolidated primitive candidates, where raw labels with the same underlying meaning had been merged, before the Cycle III freeze review.

actually contained several different semantic structures.

The need for decomposition is illustrated by the full original control-setting requirement in Figure 4.9. The full requirement is shown first so the decomposition example can be read against the complete source artifact. In Cycle II, this requirement block was treated as one compound source artifact for analysis.



Normal Operation with Constraints

If Signal_A is in the range 0% to 100% Signal_B shall be set to the value of Signal_A.

If Signal_A is a OutsideRange_A (outside the range 0% to 100%) Signal_B shall be set to 100%

If Signal_A is below the range 0% to 100% Signal_B shall be set to 0%

Fallback Functionality

If Signal_A is missing, "Status_A" or "Status_B" the following order of resolution shall be used. The first condition met shall be sufficient.

- The last known valid value (0 to 100) of Signal_A.
- Value from the following table depending on ModeTable_A:

Mode_A	Signal_B
State_A	Value_A
State_B	Value_A
State_C	Value_B
State_D	Value_B
State_E	Value_B
State_F	Value_B
State_G	Value_B

Data from the table must not be saved as last known valid value. The table is to be used as a last resort in case there is a fallback condition involving ControlledValue_A.

Figure 4.9: Full original control-setting requirement used as the source artifact in the Cycle II decomposition example.

Figure 4.10 shows how the full control-setting requirement in Figure 4.9 was decomposed into smaller semantic units for primitive discovery. The decomposition is an analytic extraction of the source artifact, not a verbatim restatement. In Cycle II, this source artifact was decomposed into five atomic statements.

1. When Signal_A ≥ 0 AND Signal_A ≤ 100, then Signal_B shall be set to Signal_A.
2. When Signal_A > 100, then Signal_B shall be set to 100.
3. When Signal_A < 0, then Signal_B shall be set to 0.
4. When Signal_A is within Invalid Signal Range, then Signal_B shall be set to last valid value of Signal_A.
5. When no valid value of Signal_A exists, then Signal_B shall be set according to Table_A.

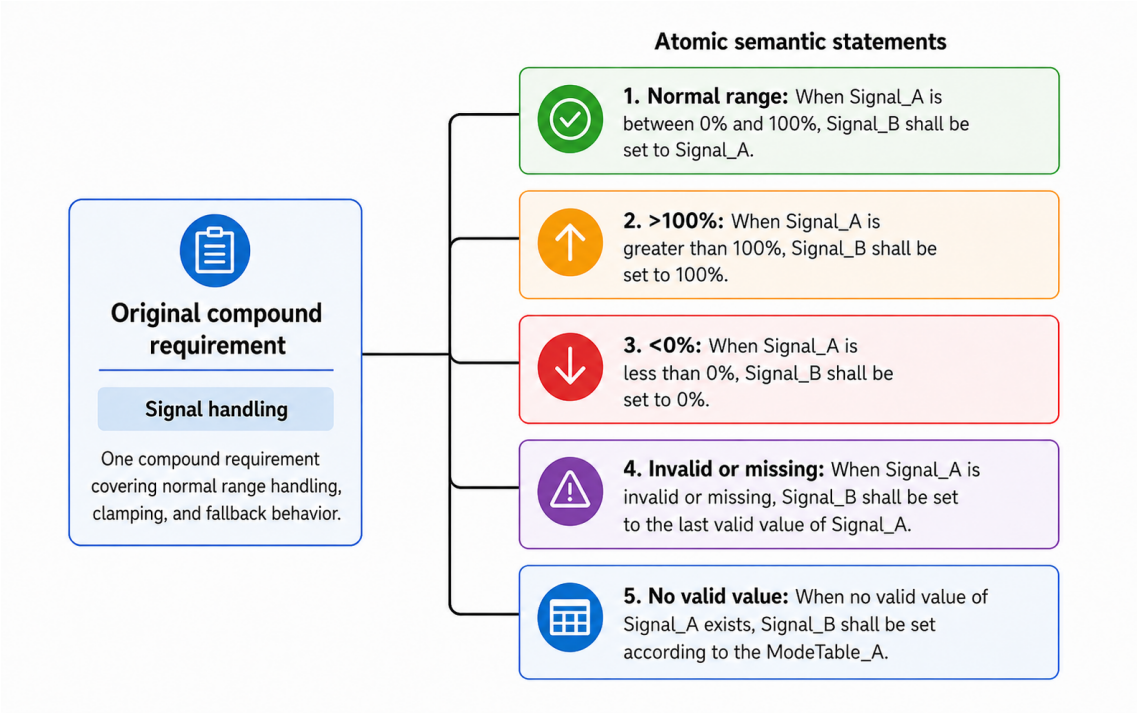


Figure 4.10: Example of atomic decomposition in Cycle II. The control-setting requirement in Figure 4.9 was treated as one compound source artifact and then decomposed into five atomic semantic statements.

This example shows why broad class labels alone were not enough. The same source artifact contained conditional assignment, a fallback where an invalid source is replaced by its last valid value, and a table-based fallback. The full requirement in Figure 4.9 also includes supporting table detail and a final note that the table value is a last resort and must not be stored as the last known valid value. That supporting context is shown in the full requirement figure but was not separated into an extra atomic statement here. It also shows how one source requirement could point toward several later primitive families.

4.2.2.1 Discovery Protocol

Primitive discovery was done class by class. For each requirement, the analysis recorded whether it contained one or more semantic statements, the decomposed atomic statement or statements, the role of each statement, a candidate primitive label, and notes on ambiguity, incompleteness, or outside-scope content. This made the discovery work traceable at the level of atomic meaning instead of whole requirements, as illustrated by the control-setting decomposition in Figure 4.10.

4.2.2.2 Batch-Based Discovery

After each batch, newly observed primitive labels were recorded. These labels were different from the broad classes from Cycle I. They captured recurring semantic units inside atomic statements. Discovery was performed in mixed-ECU batches of

about 30 requirements. A typical batch included 11 requirements from ECU_1, 13 from ECU_2, 4 from ECU_3, 1 from ECU_4, and 1 from ECU_5. This mixed-ECU design reduced the risk that the primitive labels would reflect only one ECU's wording habits. Each new atomic statement was compared with the existing primitive candidates by a constant comparative logic [29]. If a batch introduced a new semantic pattern, a new candidate was added and the next batch was reviewed. Discovery continued until semantic saturation was reached inside each broad class. In practice, this meant that a new batch did not introduce a new semantic pattern. One extra confirmation batch was then reviewed after the first zero-new batch to reduce the risk of stopping too early.

The candidate primitive labels emerged from the atomic statements, not from the broad class names. This was important because the same broad class could still contain several distinct recurring behaviors. The result of this work was 86 candidate primitive labels across the reviewed classes and ECUs.

4.2.2.3 Representative Discovery Examples

Three examples illustrate how Cycle II moved from atomic statement to primitive label. The first example came from a direct passthrough statement. The observed atomic structure was a direct assignment from one signal to another without an extra guard or transformation. This recurring structure was grouped under an unconditional assignment or passthrough primitive label. It counted as a distinct label because it expressed simple forwarding behavior that did not depend on a condition, a stored value, or a state transition.

The second example came from a guarded state-change requirement. The observed structure described a state transition that was triggered by a signal-based condition and that moved the system into a named state. This was grouped under a guarded named-state transition, signal-based primitive label. It counted as a distinct label because the requirement was not just setting an output value. It described state progression and therefore pointed beyond ordinary behavioral assignment.

The third example came from a persistent storage requirement. The observed structure described a write or update to a retained value under a guard condition. This was grouped under a guarded persistent store primitive label. It counted as a distinct label because persistence and lifecycle semantics were part of the meaning. A normal assignment label would have hidden that difference.

Taken together, these examples show that Cycle II was already discovering semantic units across what later became Behavioral, Workflow / Stateful, and Storage / Lifecycle areas. At this stage, however, the work remained a discovery activity. The final layer system was still to be refined and frozen in Cycle III.

4.2.2.4 Consolidation

After the raw discovery phase, the candidate labels were consolidated so that repeated structures would not be counted as separate primitives when they had the same

meaning but different wording. This step reduced the set from 86 candidate primitive labels to 36 consolidated primitive candidates.

One consolidation decision involved lookup and linearization behavior. Early in the review, linearization and lookup conversion appeared as separate candidate labels because some requirements used tables while others used explicit conversion wording. During consolidation, these were merged into conversion assignment because the underlying semantic role was the same, and table form alone was not enough to justify a separate primitive when the requirement still expressed a conversion assignment.

Another consolidation decision involved retained fallback behavior. At first, several retained fallback variants appeared to be separate because some requirements restored last valid value, some restored a retained value, and some used fallback tables after retained-value failure. These were kept under one fallback-oriented primitive family with variants when the core semantic role stayed the same. This showed that fallback source and fallback sequence could vary without always creating a new primitive.

Arithmetic-like derived metrics were also consolidated. Some candidate labels described derived ratio calculations, while others described average or aggregate metrics. These were absorbed into arithmetic assignment when the requirement still expressed a computed assignment from known inputs. Arithmetic variation alone did not justify separate primitive labels unless the execution meaning also changed.

Some candidates were removed from the candidate set because they were too compound to keep as standalone primitives. Multi-step workflow sequencing first appeared as its own candidate because the source wording made it look like one recurring structure. During consolidation it was dropped and decomposed because its parts were already covered elsewhere: guarded state transitions and timeout-driven branches under the Workflow / Stateful family, and behavioral assignments under the Behavioral layer. A similar decomposition applied to request and response workflow cases. These were also dropped as standalone candidates because their parts were already covered elsewhere, mainly as timeout-driven branches, guarded state transitions, and behavioral assignments.

One candidate was reclassified after closer semantic review. Event-received immediate assignment first looked like a separate stateful primitive because it referred to an event. During consolidation it was reclassified as Behavioral when the event only acted as an ordinary trigger for an immediate assignment and no real state retention or workflow progression was involved. This became an important boundary for later cycles. Event language alone did not automatically make a requirement stateful. Together, these consolidation decisions reduced the candidate set from 86 raw labels to 36 consolidated primitive candidates that were ready for the freeze review in Cycle III.

4.2.3 Evaluation

Cycle II was evaluated through saturation tracking, per-class review, and mixed-ECU credibility checks. The goal was to determine whether the discovered primitive labels

were stabilizing and whether the candidate set remained usable across several ECU contexts.

4.2.3.1 Saturation Logic

Semantic saturation was not equally clear across the broad classes. In Cycle II, the question was whether new atomic statements inside each class continued to reveal new primitive meanings. Saturation meant that additional review no longer revealed new candidate primitive meanings inside the class. Consolidation was a separate step. It meant reviewing the candidate labels to decide whether some of them expressed the same underlying meaning and should therefore be merged. Inside unconditional assignment, saturation was reached quickly because the recurring semantic units repeated with little variation. Inside conditional assignment, saturation was reached more gradually because the conditions, fallback forms, and constraint forms varied more. Algorithmic computation reached candidate saturation earlier than expected, but it remained hard to consolidate because the requirements used many surface forms and often mixed computed assignment with other behaviors.

4.2.3.2 Per-Class Discovery

The later CNL design was based on the primitive meanings discovered inside and across these classes.

Cycle II did not revisit all nine Cycle I classes with the same depth. Primitive discovery focused mainly on the classes that produced recurring semantic behavior suitable for decomposition and later CNL design. The remaining classes were reviewed more lightly and either mapped to primitive groups already covered elsewhere or were treated as outside the intended scope.

The starting classes differed in primitive yield, saturation behavior, and how clearly they mapped to later semantic layers. Conditional and unconditional assignment produced the most primitives. Algorithmic computation produced fewer stable primitive candidates and was harder to consolidate. State-machine and timed behavior mapped most clearly to later Workflow / Stateful formalization. Settings and storage management mapped most clearly to later Storage / Lifecycle formalization.

These two classes produced many recurring primitive labels because the reviewed requirements contained many assignment-style requirements, signal guards, fallback cases, and constraint variants. Unconditional assignment saturated quickly because most statements reduced to direct passthrough or direct value setting. Conditional assignment took longer because the same broad family covered ordinary guarded assignment, explicit fallback, fallback to the last valid source value, and table-driven fallback.

Algorithmic computation was the messiest class. It produced fewer stable primitive labels, but the reviewed statements were harder to translate into a controlled form. Some examples were ordinary arithmetic assignment. Others mixed arithmetic with conversion, lookup behavior, fallback rules, or state conditions. This class already hinted that one flat grammar would not be enough.

State-machine and timed behavior pushed the work toward later Workflow / Stateful formalization. These requirements often described transition rules, order dependencies, waiting behavior, or progress through named states.

Settings and storage management also produced a medium primitive yield and saturated fairly quickly. These requirements often depended on persistence, retained values, initialization, or write conditions, which is why they pushed the work toward later Storage / Lifecycle formalization.

4.2.3.3 Cross-ECU Credibility

The mixed-ECU batch design strengthened the credibility of the discovered primitive labels. Primitive discovery batches already combined several ECUs during the discovery work instead of using extra ECUs only as a late check. This reduced the chance that the resulting set would capture only one ECU's preferred wording style. The final frozen primitive set is reported in Table 4.2, summarized in Appendix B, and specified in detail in Appendices D, E, and F.

4.3 Cycle III Findings

4.3.1 Problem Investigation

Cycle III shifted from primitive discovery to final formalization of the software-level CNL. Cycle II had already produced a useful set of candidate primitive labels, but that set still belonged to the discovery phase. The remaining work in Cycle III was to determine which candidates had stable meaning, which ones could be rewritten into one consistent CNL form without changing the intended meaning, and which ones actually belonged to the same primitive after closer review.

The main issue for the CNL design was that the supported requirements did not all need the same kind of language construct. Some requirements described direct condition-action behavior that could be expressed as ordinary runtime logic over current inputs. Others required the language to express retained state, timer progress, or named workflow position. Others required the language to express persistence, initialization, or lifecycle-triggered read and write behavior. A single flat CNL form would have hidden these differences and made the language look more uniform than the underlying semantics allowed.

This design problem became visible when similar source wording produced different semantic needs in the CNL. An assignment-style sentence could still require a stateful or storage-oriented construct if its meaning depended on the workflow state already held by the system at that runtime step, on timeout expiry, or on stored values. Conversely, a sentence with event wording could still map to a simple Behavioral assignment if the event only acted as an immediate trigger. Surface wording was therefore not enough to define primitive boundaries or CNL patterns. In Cycle III, each candidate had to be tested against repeated examples to determine whether it supported one stable CNL form or whether the language needed a separate construct to preserve the intended meaning.

An example illustrates this distinction.

> "if Signal_A = State_A and Signal_B is above Threshold_A, set Signal_C to State_B"

In this example, Signal_A and Signal_B are current inputs and Signal_C is the assigned output. State_A and State_B are values used in the condition and assignment. The rule can be expressed as a direct Behavioral construct because it only depends on values available at that runtime step.

> "if GuardSignal_A is active and StateCarrier_A already held by the system is State_A, move StateCarrier_A to State_B"

In the second example, GuardSignal_A is a current input guard. The difference is StateCarrier_A. It is retained workflow state held by the system across runtime steps. This cannot be treated as ordinary assignment because the rule checks retained workflow state and then updates that same retained state. Although both examples can be written in assignment-like language, they need different CNL constructs if the formalization is to preserve the intended meaning.

The starting point for this work was the 36 consolidated primitive candidates from Cycle II. Some of these candidates already showed a consistent meaning across examples and required only limited refinement before they could be frozen as primitives. Others were still fragmented, because several candidate labels were being used for cases that later proved to express the same underlying primitive. Others looked stable until a broader review showed that their examples did not fit within one semantic layer, or that they depended on meanings the CNL did not support well enough to formalize reliably. Cycle III therefore moved from discovery to freeze testing. The main question was which semantic patterns could be expressed in the CNL without removing distinctions that mattered in the source requirements, adding information that the original requirement did not actually state, or claiming support for meanings that the CNL could not represent reliably.

4.3.2 Artifact Development

4.3.2.1 From Candidate Set to Frozen Primitive Set

Cycle III reviewed the 36 consolidated candidates from Cycle II one by one to determine whether each candidate was ready to be frozen into CNL v2. Here, frozen means that the candidate stopped being provisional and was accepted into the final supported primitive set with a defined CNL form and semantic layer. For each candidate, the review brought together the requirement examples that had been assigned to that candidate in Cycle II and compared them across repeated cases and ECU contexts. The review then asked three questions. It asked whether those examples really expressed one stable meaning, whether that meaning could be written in one consistent CNL form without adding information that was not stated, and whether the candidate belonged clearly to one of the three semantic layers of the CNL. A candidate was only frozen when repeated examples satisfied all three conditions. If a candidate kept changing definition during review, if its examples combined more than one kind of underlying behavior, or if formalizing it required interpretation that was not supported clearly enough by the requirement text, the grammar did not include it. The difference from Cycle II was that the goal was no longer to discover or consolidate candidate labels, but to decide whether the remaining candidates could be formalized as stable CNL primitives.

One recurring finding from the freeze review concerned retained fallback behavior. Early candidate labels treated three cases separately. These were cases where the output simply kept its previous value, cases where the output kept its last valid value, and cases where a condition explicitly triggered that hold behavior. At first these looked different because the source wording referred to last value, last valid value, or condition-triggered retention. Closer review showed that these cases shared the same core mechanism. When the stated condition was met, the output kept the value it already had instead of taking a new input value. The trigger condition and the exact wording used for retention could vary, but the effect remained the same and applied only to that output. These cases were therefore merged into one broader retained-fallback primitive. The final primitive allowed limited wording variants for how retention was expressed and whether an explicit guard condition was stated.

Another recurring finding from the freeze review concerned requirements that transformed a source value before assigning it. Some requirements said that a signal should be converted using a named table. Others said that a value should be linearized according to a table. Others expressed the same step inside a guarded assignment. At first these appeared as separate candidates because the requirement texts used different verbs and sentence structures. During Cycle III they were merged into one conversion-assignment primitive because the underlying meaning was the same. A source value was transformed through a named conversion reference before assignment. What varied was how the requirement text described that step, or whether it appeared on its own or inside a larger conditional statement, not the primitive itself.

State-transition candidates also appeared in several different textual forms that had to be compared and merged. Some source artifacts expressed transitions through explicit state-machine nodes and transitions. Others expressed them through state-carrier signals that were checked and then assigned a new state value. These looked different in the source system, but the semantic mechanism was the same. The state already held by the system at that runtime step, together with one or more guard conditions, determined whether the system moved to a new named state. Cycle III therefore grouped these cases under one guarded state-transition primitive with different source-carrier forms.

Timeout-related candidates showed a similar pattern. Some requirements used threshold comparisons on timers. Others used phrasing such as “has expired” or “has not been received within” a given duration. These forms first appeared as several candidates because the wording and trigger shape varied. The later review showed that these examples expressed one consistent mechanism across the reviewed cases. In each case, a branch was selected when a timeout expired. Cycle III did not keep the different timeout expressions as separate candidates. It refined them into one timeout-driven branch primitive with a small set of condition variants for the different timeout formulations.

Candidates with tables were also merged during Cycle III. Early labels distinguished passthrough tables, conditional mapping tables, priority tables, and multi-output tables. Closer review showed that these forms shared the same core mechanism. Input columns or conditions selected one or more output values through row matching. Priority ordering and the number of outputs became mode or structure parameters inside the same primitive family.

Cycle III also removed or demoted several candidates. Some were too compound and decomposed into smaller primitives. Multi-step workflow sequencing and request-and-response correlation were dropped as standalone primitives during consolidation. Other candidates were kept outside the frozen grammar because they lacked enough repeated evidence, had ambiguous wording, or required unsupported arithmetic or policy semantics.

Figure 4.11 summarizes this consolidation step. The freeze review reduced the candidate list and tested whether each accepted candidate had one consistent meaning across the reviewed examples and one clear layer fit.

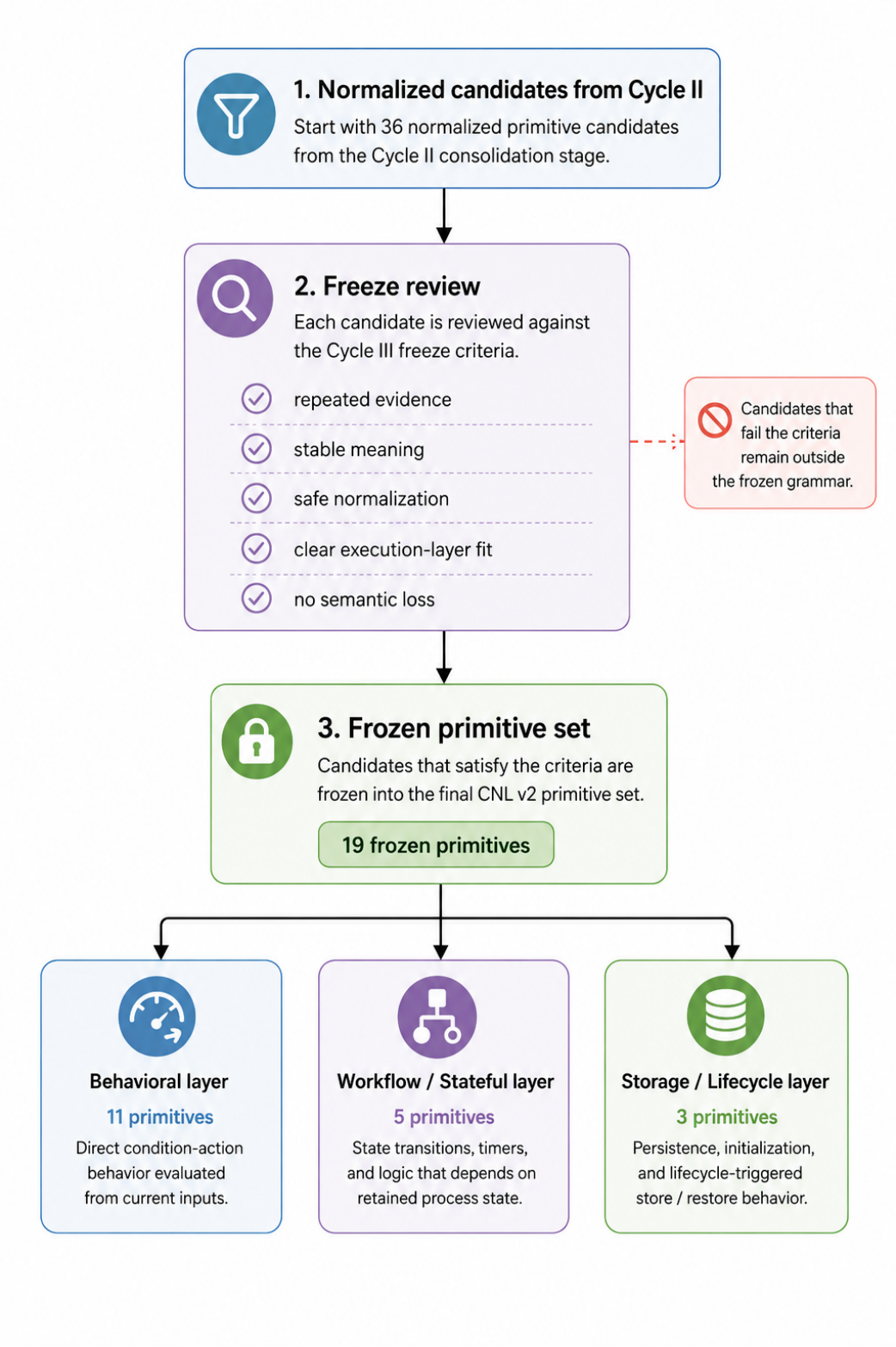


Figure 4.11: From consolidated candidates to frozen layered primitive set.

4.3.2.2 Why One Layer Was Not Enough

The freeze review showed that one flat CNL would not be enough even after candidate consolidation. The accepted candidates did not all mean the same kind of runtime behavior. Some could be evaluated directly from the input values available at the runtime step being evaluated. Some depended on the workflow state already held by the system at that runtime step or on timer progress. Others depended on storing or restoring values across lifecycle boundaries. A flat CNL would hide these differences and force one execution meaning onto primitives that work in different ways.

A Behavioral example shows the simplest case. One requirement stated that when Signal_A = State_A and Signal_B was above Threshold_A, Signal_C should be set to State_B. Additional clauses handled other ranges and corresponding outcomes. This requirement only depended on the signal values available at that runtime step. At each runtime cycle, the system read the available signal values, checked the conditions, and assigned the output.

A Workflow / Stateful example shows a different mechanism. One requirement stated that when GuardSignal_A was active and StateCarrier_A was already in one named state, StateCarrier_A should move to another named state. GuardSignal_A is a current input guard. StateCarrier_A is retained workflow state. This could not be reduced to ordinary assignment because the same state carrier was part of both the guard and the effect. The decision therefore depended on a state value that the system had retained from earlier runtime steps and then used to determine the next state.

A Storage / Lifecycle example shows a third mechanism. One requirement stated that the last valid value of Signal_A should be stored in StoredSignal_A when LifecycleEvent_A occurred, and that Signal_A should later be initialized from StoredSignal_A. This case was different from both ordinary assignment and workflow-state progression. It described a durable write at one lifecycle point and a restore or initialization action at another. The requirement therefore depended on values being preserved across lifecycle boundaries and later read back during initialization.

Figure 4.12 summarizes this split. The same overall CNL grammar could still be used across all three layers, but the parsed statements could not all be given the same semantic treatment after parsing.

4.3.2.3 Layered CNL v2 Formalization

The result of Cycle III was the layered structure of CNL v2 shown in Figure 4.12. The first layer was Behavioral. Behavioral primitives describe direct runtime behavior such as condition-action logic, value assignment, explicit fallback, priority evaluation, and other effects that can be evaluated from the input values available during an ordinary runtime check.

The second layer was Workflow / Stateful. Workflow / Stateful primitives describe state transitions, timer lifecycle behavior, timeout-driven branching, state-node output logic, and other logic that depends on remembered workflow position or other retained state. In this layer the system may need to read the state it already holds

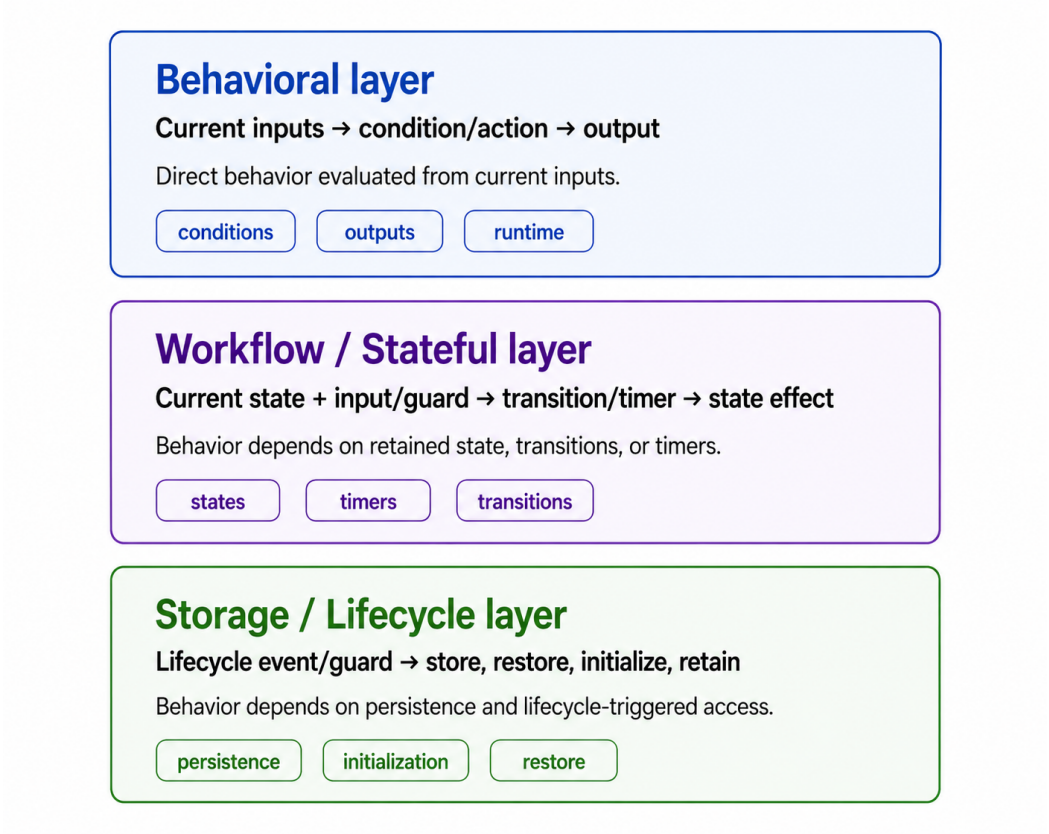


Figure 4.12: Cycle III three-layer execution split across Behavioral, Workflow / Stateful, and Storage / Lifecycle semantics.

at a runtime step, advance a timer, or wait for a duration to elapse before an effect can happen.

The third layer was Storage / Lifecycle. Storage / Lifecycle primitives describe retained values, persistence, initialization, guarded store behavior, and lifecycle-triggered read or write operations. In this layer the key meaning is that values survive across mode or power boundaries or are loaded during initialization, not just how the system behaves at a single runtime step.

This layered structure did not mean that the thesis introduced three separate languages. The controlled language still has one overall grammar. A single CNL parser could therefore recognize supported statements from all three layers. What changes after parsing is how the parsed statement must be interpreted. The resulting representation must keep track of whether a statement should be treated as direct behavior, stateful workflow logic, or storage and lifecycle logic. The interpretation rules do that work. They make explicit how each parsed statement should be understood.

The layer structure also gave a clearer rule for inclusion. A primitive could only be frozen if it fit fully within one layer. If a candidate kept shifting between layers depending on example, that was evidence that the candidate was too broad, too ambiguous, or not yet ready to freeze. The resulting three-layer structure then determined how the surviving candidates were grouped into the final frozen primitive set.

4.3.2.4 Representative Frozen Primitive Groups

The final primitive set was not evenly distributed across the three layers. Behavioral primitives formed the largest group because much of the reviewed software-level material still expressed direct runtime behavior, while the smaller Workflow / Stateful and Storage / Lifecycle layers captured the cases that could not be reduced to simple assignment logic without losing important meaning. Table 4.2 shows the full grouped primitive set that remained after consolidation and layer testing.

Layer	**Frozen primitive groups**
Behavioral	Unconditional assignment, simple conditional assignment and explicit fallback, validity-gated assignment, target-local retained fallback, multi-assignment with AND, event-received immediate assignment, conversion assignment, source-memory fallback, table-driven assignment, priority-ordered conditional assignment, arithmetic assignment (partial)
Workflow / Stateful	Guarded named-state transition, timer lifecycle control, timeout-driven branch, state-node output logic, incremental accumulator
Storage / Lifecycle	Lifecycle-triggered store/restore, stored-value initialization with default fallback, guarded persistent store

Table 4.2: Grouped frozen primitive set from Cycle III.

The most difficult primitives to freeze were fallback to the last valid source value,

incremental accumulator, and event-received immediate assignment. Fallback to the last valid source value required careful separation from target-local retention, stored persistent values, and workflow-held values. Incremental accumulator was difficult because it was the only accepted stateful computation primitive, while broader filtering and moving-window computations were excluded. Event-received immediate assignment was difficult because event wording initially suggested Workflow / Stateful semantics, but the accepted subset only used the event as an immediate trigger for Behavioral assignment.

This grouped result also explains why the final primitive set is smaller than the earlier candidate set. Cycle III did not try to preserve every discovery label from Cycle II. It reduced the language to the primitive groups that remained stable after consolidation and layer testing. The result was a clearer mapping between source meaning, controlled syntax, and how the statement should be interpreted.

4.3.2.5 Translation, Syntax, and Grammar Work

Cycle III also formalized the controlled syntax that each frozen primitive could use. This work turned repeated semantic patterns into canonical CNL forms that could be parsed reliably and checked for meaning preservation.

One example concerned fallback to the last valid source value. Source requirements used phrases such as “consider the signal to have its last known value” and “the last valid value shall be used” when a signal became invalid or missing. Cycle III translated the stable validity-triggered cases into one controlled form such as “When Source is within Invalid Signal Range, then Source shall be treated as its last valid value.” This translation removed source-specific wording differences but kept the semantic trigger explicit. It also avoided collapsing other meanings of “last known value” into the same primitive when the surrounding context suggested stored value, last received value, or another mechanism.

Figure 4.13 shows one reviewed requirement of this kind together with its translated form.

The result was more than surface cleanup. The translation fixed the trigger and the mechanism so that the primitive did not absorb storage cases or vague “last known value” cases that actually meant something else.

Another example concerned guarded persistent store behavior. Source requirements often mixed explanatory wording with the main store rule. One reviewed requirement stated that Signal_A should be stored in StoredSignal_A, and that StoredSignal_A should remain unchanged when Signal_A was missing. Cycle III translated such cases into one explicit controlled store form with a clear non-store branch. This made the persistence action visible and avoided vague wording such as “remain unchanged” when the actual logic was that no write should occur under that guard.

Figure 4.14 shows one reviewed requirement of this kind together with its translated form.

This example shows why explicit storage verbs mattered. The translated form

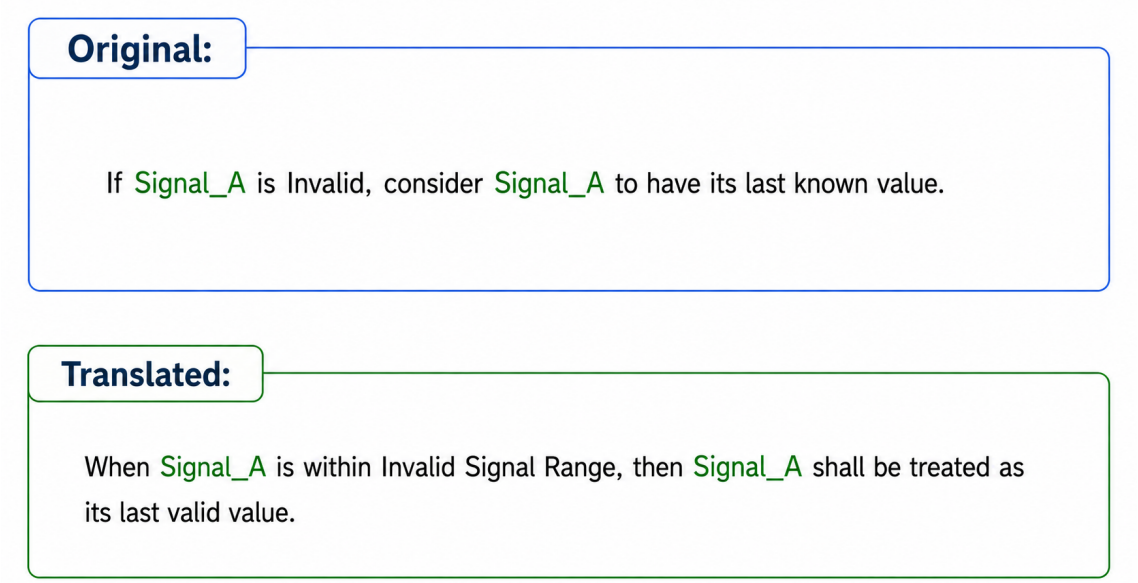


Figure 4.13: Example of translation to fallback to the last valid source value in Cycle III. The figure shows one original requirement and its translated form.

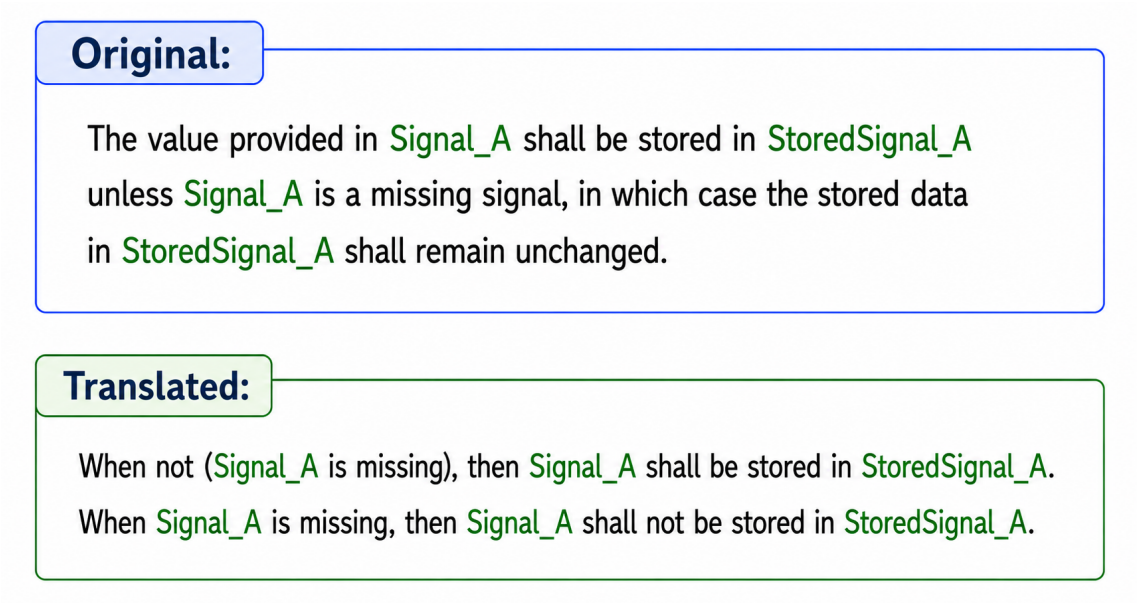


Figure 4.14: Example of guarded persistent store translation in Cycle III. The figure shows one original requirement and its translated form.

exposed the actual action. Either a store occurred or it did not occur. The final syntax therefore made persistence behavior easier to interpret, easier to compare across rows, and easier to separate from ordinary fallback logic.

Priority-ordered conditionals also required explicit syntax. In the source material, textual order often implied evaluation order without stating it directly. Cycle III formalized this by requiring explicit priority marking when the result depended on first-match evaluation. A source set of five guarded assignments could therefore be rewritten as a priority-ordered series of CNL statements. This made the evaluation order visible to both readers and the parser.

This change mattered because the source rows often looked like a list of independent conditions when they were actually an ordered decision chain. Without explicit priority syntax, the parser could still read the sentences, but the execution meaning would remain under-specified. Cycle III therefore included ordering in the primitive meaning itself.

Event-received immediate assignment provided a fourth example. Some source requirements used compact record-style assignment inside one sentence after an event was received. Cycle III expanded these into explicit field-level assignments joined by AND. This avoided introducing a separate record-literal grammar construct and kept the behavior within the frozen Behavioral layer when the event only acted as an immediate trigger and did not imply retained workflow state.

Figure 4.15 shows one reviewed requirement of this kind together with its translated form.

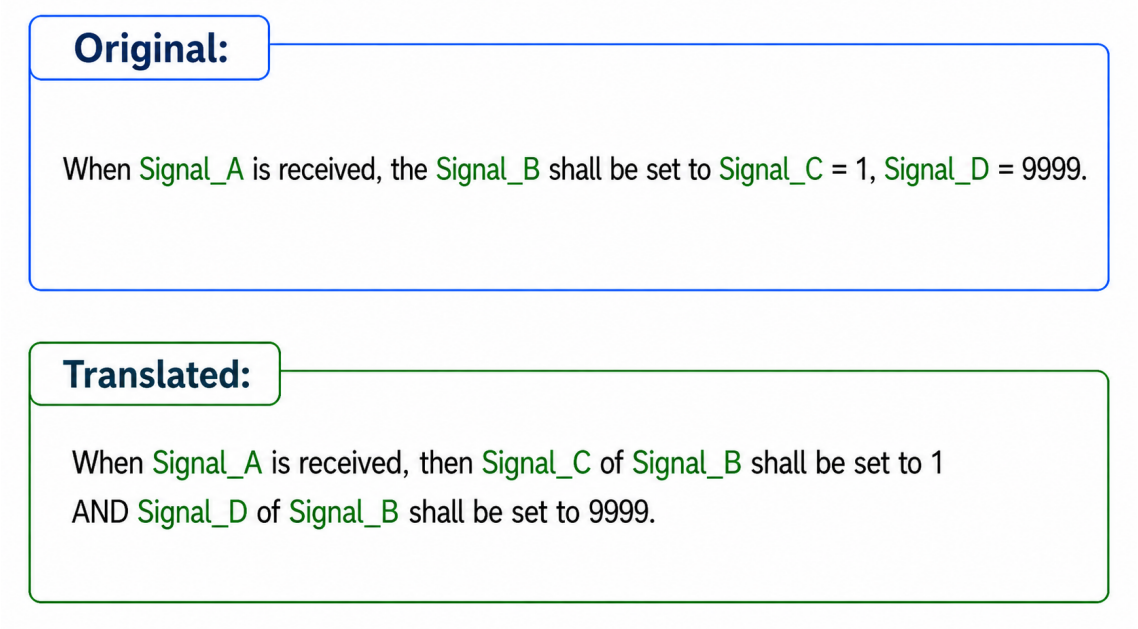


Figure 4.15: Example of event-received immediate assignment translation in Cycle III. The figure shows one original requirement and its translated form.

This translation choice let Cycle III reuse the existing assignment pattern instead

of adding a new special form to the grammar. It also stated each changed field separately, which improved both readability and parse consistency.

The grammar work during Cycle III used these controlled forms to define the final language basis. This followed the general CNL idea of restricting natural language to improve machine interpretability while keeping readability [3]. The syntax was specified in EBNF-style form and implemented with Lark [12]. Valid statements could be parsed into a tree and then transformed into an intermediate representation that exposed the elements needed for later interpretation. The key result in Cycle III was that the grammar and the translation rules were tied to frozen primitives, and that each primitive had a clear place in one layer.

4.3.3 Evaluation

4.3.3.1 Evaluation and Freeze Logic

Across the Cycle III review material drawn from the software-level requirement datasets, a total of 263 requirements were assigned primitive labels during the later semantic review that froze the final primitive set. At the start of Cycle III, the review worked from the 36 consolidated candidate primitive labels produced in Cycle II, and the freeze step later reduced these to the 19 frozen primitive groups shown in Table 4.2. This review was the main evaluation step for Cycle III. Unlike the earlier reviews in Cycle I and Cycle II, which focused on discovering broad classes and then candidate primitives, this review tested whether the surviving candidates were precise enough, consistent enough across examples, and well supported enough to be frozen into the final CNL. Its purpose was therefore to test whether each candidate primitive had enough repeated evidence, one consistent meaning across the reviewed examples, one consistent CNL form that did not change the intended meaning or invent missing information, and a clear fit in one layer.

The first freeze criterion was repeated evidence. A candidate was not frozen because it looked plausible in one or two examples. It had to appear across multiple requirements, and preferably across several ECU contexts, without changing its core meaning. This reduced the risk that the grammar would freeze one team's local wording habit as if it were a general semantic primitive.

The second criterion was one stable meaning. Every accepted instance of a frozen primitive had to reduce to the same core mechanism. If a candidate kept producing cases that required a wider definition, extra hidden assumptions, or a different interpretation, this counted as instability. Stable primitives did not grow during review. New examples either matched the fixed definition or fell outside it.

The freeze review still used constant-comparative logic, but for a different purpose than primitive discovery. Each new instance was compared with the current candidate definition to decide whether it matched that definition, showed that the definition needed refinement, or showed that the candidate should not be frozen as one primitive [29]. This moved the work from discovering possible candidates toward deciding which candidates were stable enough to keep in the final primitive set [30].

Stability was judged by repetition and by whether additional examples fit the primitive definition without forcing it to expand. Conversion, timeout, table-driven assignment, source-memory fallback, guarded state transition, and storage primitives remained stable when new examples were reviewed, whereas broader received-event families, request and response correlation, and ambiguous non-validity-triggered "last known value" wording did not.

The third criterion was clear layer compatibility. A frozen primitive had to fit one execution model. This was especially important in Cycle III because some candidate labels had been applied to examples that looked similar in wording but did not work in the same way. In some cases, one group of examples could be expressed as direct condition-action logic, while another group required retained state or storage behavior to preserve the intended meaning. When that happened, the candidate could not be kept as one primitive and instead had to be separated, refined, or excluded.

The fourth criterion was consistent CNL rewriting without meaning change. The question was whether a candidate could be formalized from its supporting source examples without the authors having to add signals, states, stored values, or other meaning that was not actually stated. Small wording cleanup was allowed. Sentences could be rewritten into a consistent style, and the order of steps could be made explicit when the source meaning clearly required it. What was not allowed was to make the candidate work only by adding information that was missing from the reviewed requirement examples.

The fifth criterion was no semantic loss in the canonical form. A primitive was only frozen when the translated CNL sentence still preserved the meaning needed for parsing and validation. If an important distinction disappeared during translation, the candidate was too broad or too unsafe to freeze in its current form.

These criteria also gave a clear basis for exclusion. A candidate was not frozen when examples showed inconsistent meaning across cases, when the reviewed rows decomposed into smaller existing primitives, when the evidence was too weak, when the boundary against nearby cases was too vague, or when the candidate required unsupported execution behavior such as more open-ended arithmetic, protocol-level control logic, or undefined storage semantics.

The later semantic review showed that the frozen primitives had stable boundaries. Repeated examples did not force the accepted definitions to expand. Conversion cases stayed conversion cases after translation into a controlled form. Timeout cases stayed timeout cases even when the timeout wording varied. Guarded state-transition cases still depended on the state already held by the system at the runtime step together with guard logic, not on simple assignment semantics. Storage cases remained storage cases because their core meaning was persistence or lifecycle-triggered access.

Some primitive families were especially important in this review. Source-memory fallback needed close checking because similar last-value wording appeared in Behavioral, Workflow / Stateful, and Storage / Lifecycle contexts. Table-driven assignment also needed close checking because tables could express simple mapping, conditional

mapping, or priority logic. The later review showed that table meaning could be stabilized when the evaluation mechanism stayed explicit, while more ambiguous table-property cases had to stay outside the frozen grammar.

This review also strengthened the credibility of the final primitive set across datasets. Patterns that appeared in more than one ECU context were less likely to be artifacts of one local writing style. At the same time, the exclusion rules prevented the grammar from claiming support for every observed requirement form. This was an important evaluation result in itself because blocking translation when meaning could not be preserved was part of the artifact boundary.

4.3.3.2 Borderline and Excluded Cases

These cases were close enough to supported primitives to test the layer boundaries, but not all of them could be accepted into the frozen grammar.

Borderline cases were useful because they tested whether the layer boundaries were real or only convenient labels. One such case used ordinary condition-action wording but included a sustained-duration condition such as “for more than” a named timer. At first sight this looked like Behavioral assignment. Closer review showed that the system had to remember how long the condition had remained true before the assignment could fire. These cases therefore belonged to Workflow / Stateful logic when the duration tracking was part of the meaning.

Another borderline case looked like assignment with retained value wording but actually controlled whether a value in persistent storage should remain unchanged. The surface form suggested ordinary fallback. The semantic mechanism was persistence control. These cases therefore belonged to Storage / Lifecycle semantics.

A third important borderline involved non-validity-triggered “last known value” wording. This phrase appeared across several different contexts in the material. In some cases it meant last valid signal value. In others it could mean last received value, last stored value, or the value before shutdown. When the surrounding requirement did not make the trigger and mechanism explicit, the phrase remained ambiguous and did not become one frozen grammar primitive.

Another borderline case concerned state transitions versus state-node output logic. Some requirements described when the system should move from one named state to another, while others described which output should be produced while the system remained in a given state. These were kept separate because transition rules define movement between states, while state-node output logic defines behavior inside a state. Collapsing the second into transition logic would hide the difference between leaving a state and producing output while staying in it.

Several other frozen primitives also needed short boundary checks because their surface wording could look similar. Target-local retained fallback means that the target output keeps its own previous value. Source-memory fallback means that an invalid or missing source is treated as its last valid value before assignment. Storage / Lifecycle cases differ from both because the value is written to or restored from persistent storage. Timer lifecycle control also differs from timeout-driven branching.

The first defines when a timer is started, stopped, or reset. The second defines what happens after a timer has expired.

Some excluded candidates failed because they decomposed into smaller supported patterns. Request and response correlation is one example. The reviewed cases could be explained through a timeout-driven failure path and ordinary guarded success behavior. The correlation wording did not justify a separate frozen primitive once the underlying mechanism was decomposed.

Other excluded candidates were unstable or outside scope. Multi-step workflow sequencing did not show enough clean in-scope evidence for a stable standalone primitive. More open-ended arithmetic, filtering, and windowed computation also remained outside the frozen grammar because they required broader execution behavior than the current CNL could support within the present scope.

Two additional exclusion cases were also important. Deterministic date and time transformation cases appeared too rarely to justify freezing in the current thesis scope. Timer-expiry shorthand also remained outside the canonical syntax when the source wording used the same name for elapsed time and threshold meaning, because translation would then require invented variable distinctions that were not present in the source requirement.

4.3.3.3 Cycle III Main Result

Cycle III froze the final primitive set for CNL v2 and formalized the language as a layered design. The final frozen set contains 11 Behavioral primitives, 5 Workflow / Stateful primitives, and 3 Storage / Lifecycle primitives. The result was one grammar with clear meaning rules for each layer.

The resulting artifact is documented in several appendix materials. A compact overview of the final frozen primitive set is provided in Appendix B. The full layer-specific specifications are provided in Appendices D, E, and F. The implemented grammar basis is provided in Appendix C.

This changed the status of the CNL from an exploratory translation artifact into a stable artifact for the supported scope. The language no longer depended only on recurring surface phrasing. It now depended on frozen primitives whose meaning had been checked against repeated examples, translation rules, and layer boundaries.

Cycle III also made the exclusions explicit. Cases that were unclear, weakly supported, compound, or dependent on unsupported semantics were kept outside the frozen grammar. This gave the artifact a safer boundary and prepared the work for Cycle IV, where the frozen language was validated through survey feedback, unseen requirements, and later refinement.

4.4 Cycle IV Findings

4.4.1 Problem Investigation

Cycle IV focused on validation and refinement. After the layered CNL had been frozen, the remaining question was whether it also worked on unseen requirements and whether engineers found the translations understandable and useful. The validation therefore had to test both translatability and usability.

The main coverage question in this cycle was whether unseen material introduced new requirement semantics that the frozen primitive set could not represent. The aim was to test whether the compact primitive set from Cycle III captured the recurring requirement semantics that appeared in practice and still blocked or flagged requirements that were ambiguous, unsupported, or outside the intended scope.

4.4.2 Artifact Development

Cycle IV produced two validation artifacts. The first was a structured review sheet for unseen requirements. It recorded each reviewed requirement together with its decomposition when needed, its translatability judgment, and the primitive label or labels used in the assessment. The second was a survey with original requirement and CNL translation pairs selected to cover different primitives and translation types.

4.4.2.1 Survey-Based Usability Validation

The survey track was designed to provide direct usability feedback on generated CNL translations. It used ten selected original requirement and CNL translation pairs chosen to cover different primitives and translation types. Eleven practitioners answered the survey. The respondent group consisted of four software developers, three logical designers, two V3 testers, one V6 tester, and one system owner. Six respondents reported more than ten years of experience, four reported two to five years, and one reported zero to one year. The questions focused on readability, wording, ambiguity, and preference between the original requirement wording and the CNL translation. In this survey, unambiguity means how far the respondent judged the translation to avoid competing interpretations. Clarity means how easy the respondent judged the requirement logic and intended effect to understand from the wording. Preference means whether the respondent favored the CNL translation over the original requirement wording as a way of presenting the same content. Per-item survey results are summarized in Table 4.3. The survey instrument is provided in Appendix G.

4.4.2.2 Detailed Survey Examples

The survey used ten examples, but this thesis discusses five of them in more detail because they show the clearest lessons from Cycle IV. These five cases are used to explain both strong feedback and refinement triggers. Each case is presented with the survey figure, the item-level scores, selected participant comments, and

Item	Unambiguity (mean)	Clarity (mean)	Preference (mean)
1	9.55	9.82	9.55
2	8.64	8.45	7.45
3	7.73	8.45	6.91
4	9.27	9.45	6.73
5	8.18	8.00	4.64
6	9.27	9.18	8.00
7	9.18	9.27	8.64
8	9.00	9.18	8.55
9	8.18	7.55	6.91
10	8.36	8.64	7.09
Overall	**8.74 (med 10)**	**8.80 (med 10)**	**7.45 (med 8.5)**

Table 4.3: Per-item survey results for the ten original requirement and CNL translation pairs used in Cycle IV.

the refinement lesson it produced. The first case is given below. The remaining examples are Items 5, 6, 8, and 10, covering a table-based weak-preference case, a compact table case with positive preference, a major priority-rewrite case, and a final table-cell notation case that triggered a CNL v3 refinement.

Item 1: High ratings, but unsafe translation Figure 4.16 shows survey item 1. This example became important because the translation looked strong to respondents, but it was later re-evaluated as unsafe.

Item 1 received one of the strongest survey responses. The mean score for unambiguity was 9.55 out of 10. The mean score for clarity was 9.82 out of 10. The mean score for preference over the original was also 9.55 out of 10. Ten of the eleven respondents gave preference scores of 8 or higher. The response was therefore strongly positive overall.

The core issue appeared during re-evaluation of the translation logic. The translation had inserted `OR` between the bullet-point conditions. Two participant comments identified this directly. The later re-evaluation showed that the source could support either `AND` or `OR`. The source was ambiguous because the conjunction was not explicit.

One participant said:

> *"The translation is incorrect though, it should be "!= Active AND", not "!= Active OR"."*

Another participant wrote:

> *"1- The requirement can be interpreted in two ways, It can be AND or OR between the two statements in the bullet points, which raises the question, why it is considered as OR in the translated version."*

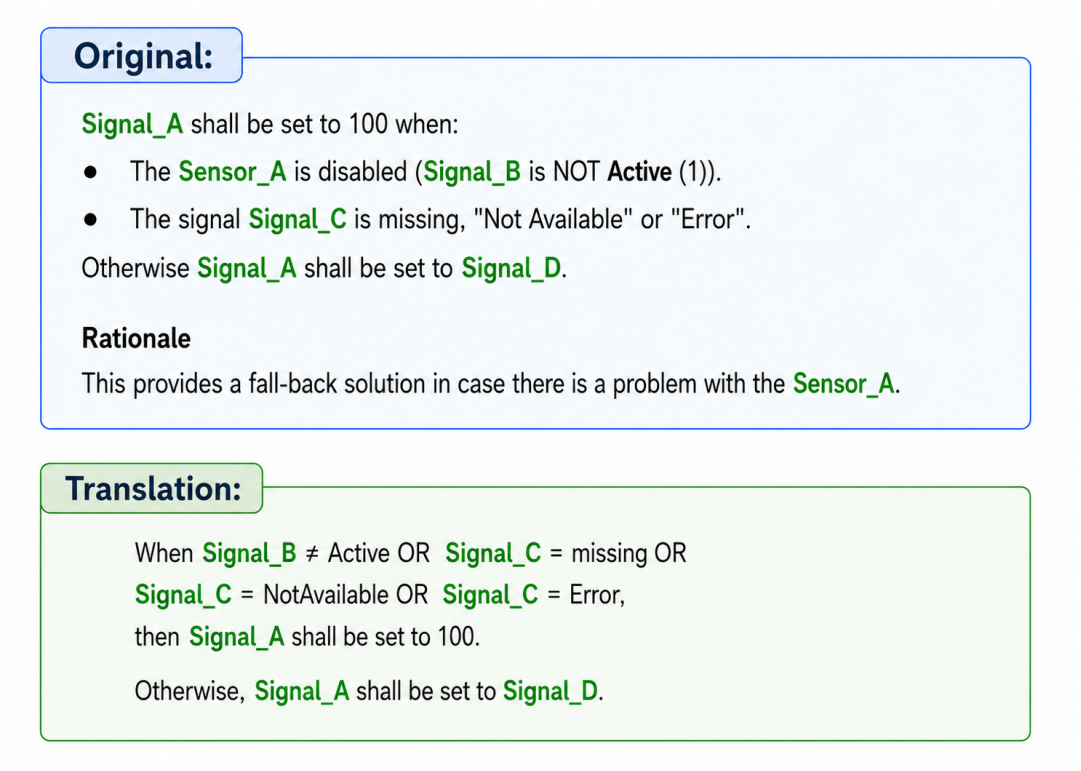


Figure 4.16: Survey item 1. The original requirement and its CNL translation received strong readability ratings, but the translation was later re-evaluated as unsafe because it added an OR that was not stated in the source.

At the same time, the positive feedback still matters. Even though the translation was later judged unsafe, respondents still rated it highly for clarity and structure. A translation can be easy to read and still be semantically unsafe.

One participant said:

> *"The CNL version has cleaner and better structure. It uses pseudocode format that I think will simplify the implementation."*

A second participant commented:

> *"The "Translation" is how a proper logical design should be written, however formatting is important so please consider that. I notice a code indentation is followed which is good."*

This case led to a stricter Cycle IV refinement rule for CNL v3. When a prose list or bullet list does not make the logical connector explicit, the requirement must remain ambiguous and must stop before translation. This follows the same boundary principle used elsewhere in the thesis. Unclear source semantics must be flagged, not invented.

Item 5: Clear translation, but table strongly preferred Figure 4.17 shows survey item 5. This example became important because it exposed the limits of flattening a compact source table into long prose-like CNL.

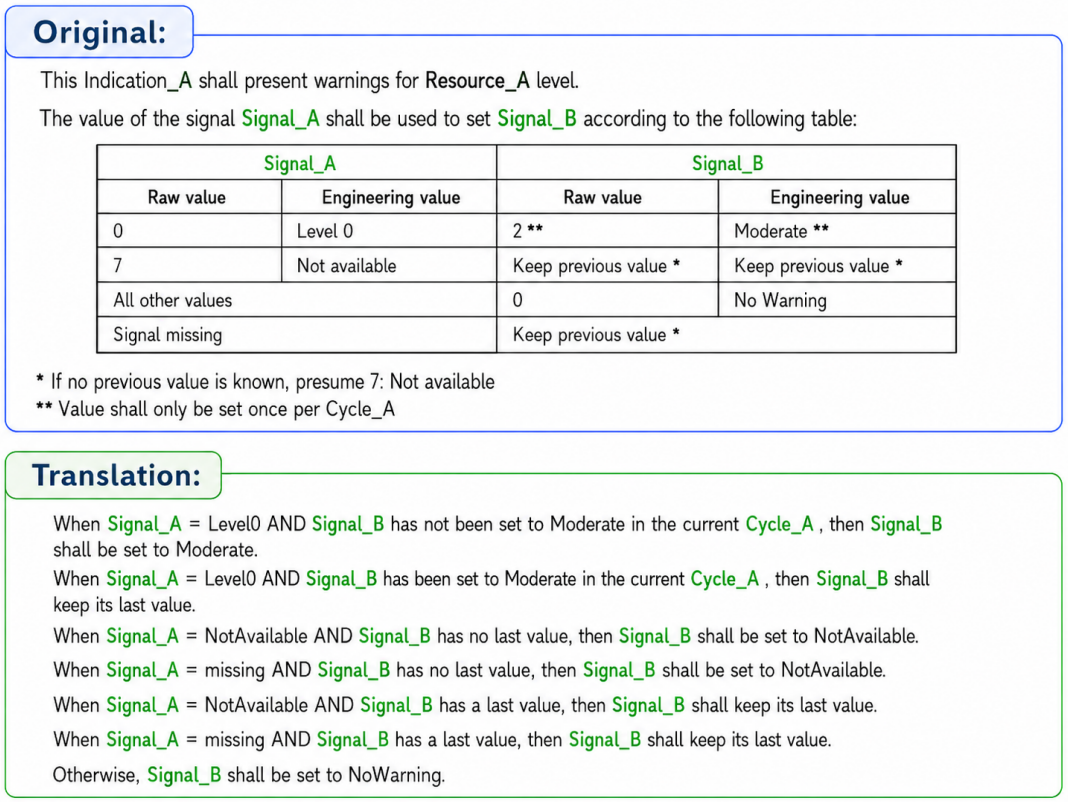


Figure 4.17: Survey item 5. The translation remained fairly clear, but the original table was often preferred because it gave a faster overview of the structured mapping.

Item 5 received a mixed response. The mean score for unambiguity was 8.18 out of 10. The mean score for clarity was 8.00 out of 10. The mean score for preference over the original was 4.64 out of 10, and the median preference score was 4. Only three of the eleven respondents gave preference scores of 8 or higher, while six gave scores of 5 or lower. The translation was therefore still judged fairly clear and mostly understandable, but preference for the CNL was the lowest among the ten survey items.

This case matters because it gave the strongest survey signal that tables can be better than flattened text when several conditions and outputs must be scanned together. The CNL made the logic explicit, but it also made the requirement longer and more repetitive. The case shows that table structure itself can carry useful readability value for engineers.

One participant said:

> *"The table has a better and quicker overview"*

Another participant wrote:

> *"I like the original requirement table format. It makes clearer structure with less text."*

A third participant commented:

> *"tables when it comes to multiple variables makes it easier to catch the pattern and the different conditions"*

The supporting comments point in the same direction. One respondent wrote that the translation was too repetitive and that a table was generally easier to read when using value tables. Another pointed out that repeating the last-value condition line by line made the translation longer than needed and made it harder to keep attention on the main pattern. Together, these comments show that the cost of full expansion grows when table rows share structure.

This also fits the earlier interview findings from Cycle I. The thesis had already found that tables could be useful for structured mappings, but that they still needed selective handling because they did not always express one simple semantic pattern. Item 5 adds a more concrete Cycle IV usability lesson. Even when a table-based requirement can be translated into valid CNL, the original source table may still be easier for engineers to scan and compare.

Source format should therefore remain a main concern in later CNL and pipeline decisions. Readability for engineers depends on semantic explicitness and on whether a useful compact representation is preserved.

Item 6: Table translated to text, but CNL still preferred Figure 4.18 shows survey item 6. This example became important because it shows a table-based requirement where the CNL remained implementation-friendly and was still preferred by most respondents.

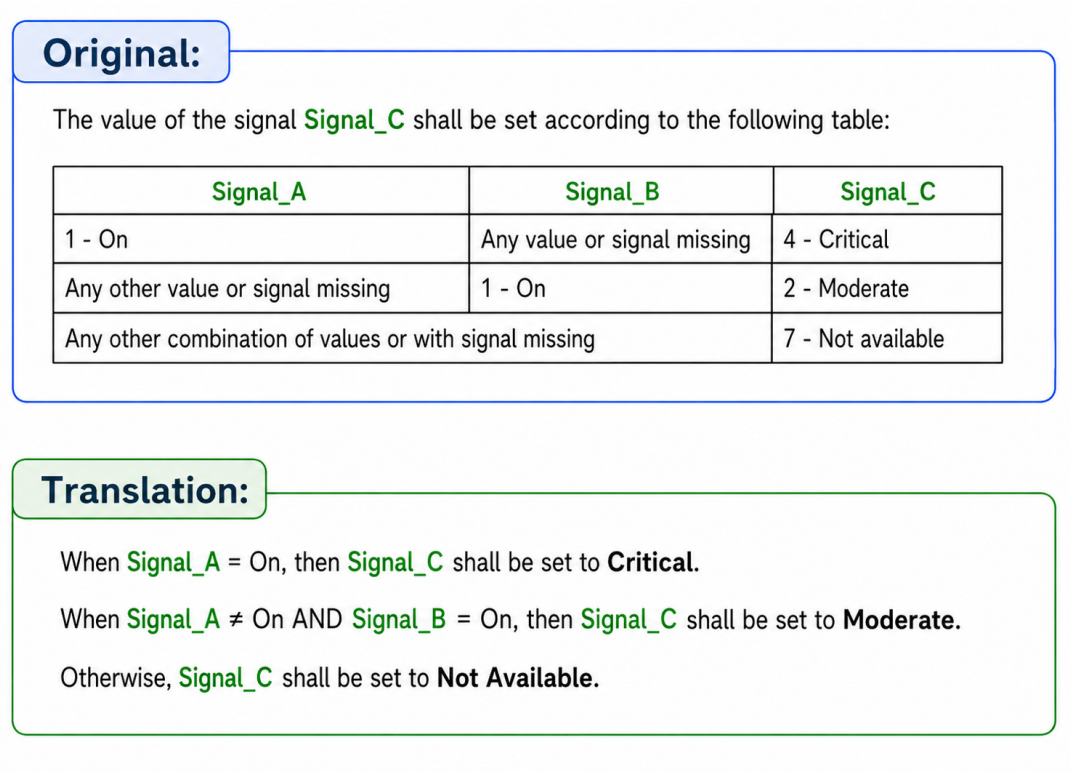


Figure 4.18: Survey item 6. The source also used a table, but this translation remained compact enough that most respondents still preferred the CNL form.

Item 6 received a clearly positive response. The mean score for unambiguity was 9.27 out of 10. The mean score for clarity was 9.18 out of 10. The mean score for preference over the original was 8.00 out of 10, and the median preference score was 10. Seven of the eleven respondents gave preference scores of 8 or higher. Nine gave scores above 5, and only two gave scores of 5 or lower. The translation was therefore judged highly unambiguous and highly clear, and preference was clearly positive overall.

Item 6 also converts a table into text, but the response here was much more positive. A plausible interpretation, supported by comparison with item 5, is that this source table was simpler to flatten because the rule set was compact and the resulting text stayed short enough to scan easily. The survey does not prove that this is the only explanation, but it does show that not all table-to-text rewrites were disliked.

One participant said:

> *"Both are equally clear not ambiguous but the CNL version is written in pseudocode format that suits better to the implementation"*

The contrast between items 5 and 6 helps refine the table-related lesson from Cycle IV. Item 5 suggested that larger or more repetitive table mappings lose readability when flattened into text. Item 6 suggests that smaller table mappings can survive translation into CNL without the same penalty. This fits the earlier item-5 comment that tables help when there are multiple variables and many conditions. The survey

instead suggests that denser tables lose more when their structure is rewritten as linear text.

Item 8: Major rewrite, but still well received Figure 4.19 shows survey item 8. This example became important because the CNL made the priority logic explicit in a much more structured way while still being preferred by most respondents.

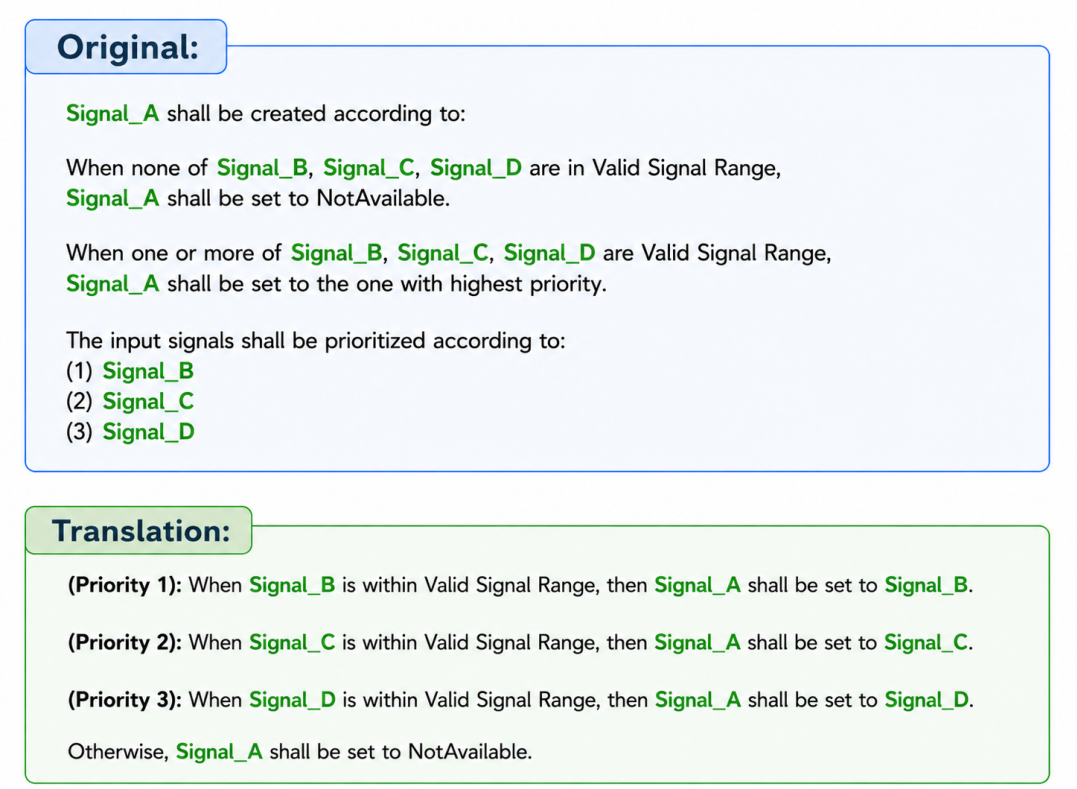


Figure 4.19: Survey item 8. The translation rewrote the source into explicit priority rules and an otherwise fallback, but the CNL was still strongly preferred.

Item 8 received strongly positive feedback across all three survey questions. The mean score for unambiguity was 9.00 out of 10. The mean score for clarity was 9.18 out of 10. The mean score for preference over the original was 8.55 out of 10, and the median preference score was 10. Nine of the eleven respondents gave preference scores of 8 or higher. Ten gave scores above 5, and only one gave a score of 5 or lower. This matters because the translation is visibly more different from the source than many of the other examples, yet it was still rated highly.

This case also helps explain why some stronger restructurings were accepted. The source used descriptive prose plus a priority list. The translation rewrote that material into explicit `(Priority N)` rules and an `Otherwise` fallback. This is a larger structural rewrite than in some of the other survey examples, but respondents still reacted positively. This fits the Cycle III result that explicit priority syntax makes the evaluation order visible.

The contrast with items 5 and 6 helps complete the table-related picture from Cycle

IV. Item 5 showed that some flattened table rewrites lose readability. Item 6 showed that compact table rewrites can still be preferred. Item 8 shows that even a larger structural rewrite can be accepted when the new form makes priority and fallback logic explicit. Readers may therefore accept a translation that looks very different from the original if it exposes the decision logic more clearly.

Item 10: Well received overall, but catch-all notation needed refinement
Figure 4.20 shows survey item 10. The translation was mostly successful, but the * catch-all notation was judged unclear and later replaced in CNL v3.

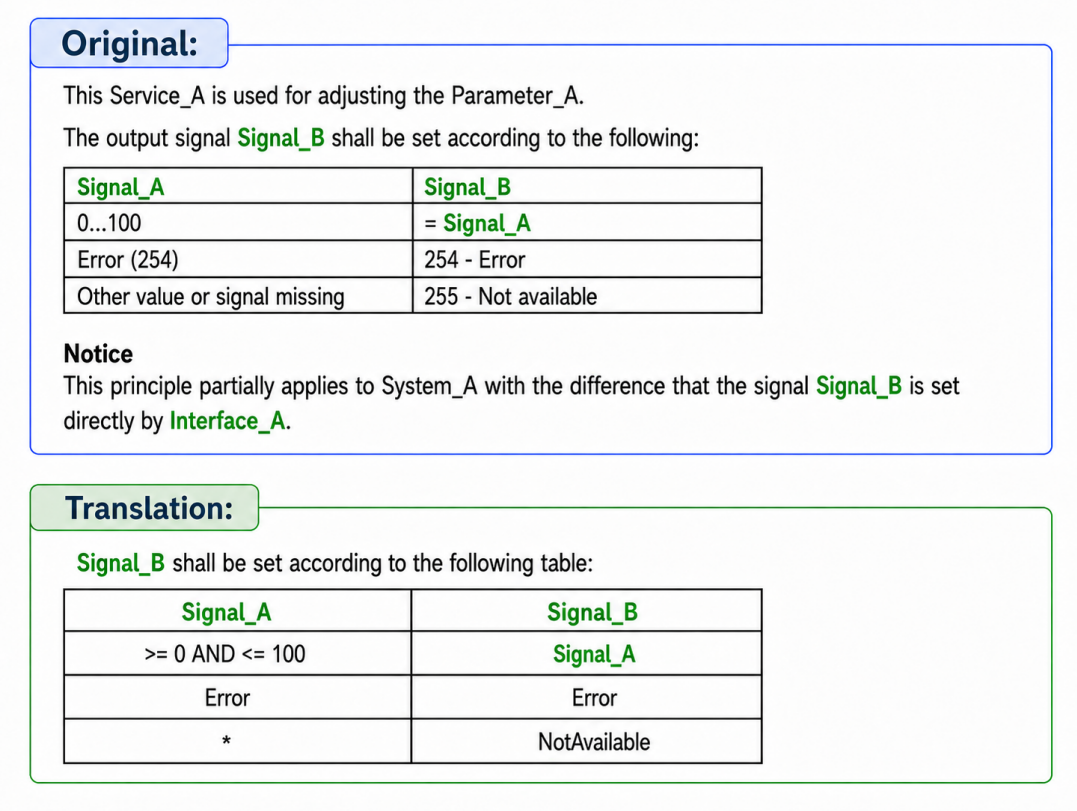


Figure 4.20: Survey item 10. The translation preserved table form and was generally well received, but the * catch-all notation exposed a visible surface problem that was later fixed in CNL v3.

Item 10 received positive feedback overall. The mean score for unambiguity was 8.36 out of 10. The mean score for clarity was 8.64 out of 10. The mean score for preference over the original was 7.09 out of 10, and the median preference score was 8. Six of the eleven respondents gave preference scores of 8 or higher. Seven gave scores above 5, and four gave scores of 5 or lower. The translation was therefore rated more positively than negatively, but less strongly than items 1, 6, and 8. This makes it a useful final case because it combines generally positive reception with a concrete notation problem.

This case matters because the source was already table-based and the translation preserved table form. The problem was not table-to-text flattening. The problem

was the use of `*` as a catch-all row marker. Respondents reacted to that notation as unclear or non-standard in logic design.

One participant said:

> *"The (*) can be misleading. Also signal missing is not a common sense or commonly understandable as it is part of the values or cases to be tested."*

Another participant wrote:

> *"* generally not used in logic design"*

The supporting comments point in the same direction. One respondent said that the original table was easier to read. Another wrote that useful information was also lost. Together, these comments show that even when table form is preserved, small notation changes can still reduce readability or remove context.

Cycle IV used this feedback to refine CNL v3. Instead of `*`, the final table catch-all wording is now chosen explicitly from `Otherwise`, `Signal missing`, or `Otherwise or signal missing`. Which label is used depends on the semantics of the source row. This was incorporated into CNL v3 and matches the final artifact rule for table-cell catch-all forms.

Even when the underlying logic is preserved, unclear notation can reduce trust and readability. This is why CNL v3 made catch-all table rows easier to read.

4.4.2.3 Survey Summary

Across the ten translation pairs, the survey responses were broadly positive on clarity and unambiguity. On the ten-point rating scale, the mean score for unambiguity was 8.74 and the median was 10. The mean score for clarity was 8.80 and the median was 10. Preference for the CNL over the original wording was more mixed, but still positive overall, with a mean of 7.45 and a median of 8.5. The results show that ratings for clarity and unambiguity were somewhat higher than ratings for preference. Respondents generally judged the CNL easier to interpret and less ambiguous than the source wording, but they did not always prefer the rewritten form.

The survey shows that the CNL improved explicitness, structure, and perceived clarity, but preference depended strongly on the type of original requirement. When the original used compact tables or familiar notation, some respondents preferred it because it was faster to scan or kept useful context in a tighter form.

The preference scores also varied by experience group. Respondents with more than ten years of experience gave a lower average preference score for the CNL, 6.80, than respondents with two to five years of experience, 8.32. The single respondent with zero to one year of experience had an average preference score of 7.80.

The strongest preference for the CNL appeared when the rewrite made the logic more explicit and closer to an implementation-friendly structure. The weakest preference appeared when compact tables were flattened, when repeated condition lines felt verbose, or when supporting context looked reduced. These tendencies are illustrated in the detailed survey examples above, especially Items 5, 6, 8, and 10. The comments

show that this lower preference did not amount to rejection of the CNL concept itself. Instead, it usually reflected specific concerns about how a given translation had been rewritten.

The qualitative comments also help separate criticism of the CNL concept from criticism of particular examples. Several comments praised the increased explicitness and the more uniform structure, but other comments pointed to concrete problems in specific cases, such as the unsafe added OR in Item 1 and the unclear catch-all notation in Item 10. These remarks matter because they show that the survey confirmed readability and also exposed translation choices that needed careful checking during refinement.

Taken together, the survey adds descriptive usability evidence for the frozen language in its intended review setting. Practitioners could review the generated translations directly and judge whether the controlled phrasing remained understandable and useful. The main result is that the CNL generally improved clarity and reduced ambiguity, while preference depended more on the local rewrite used in each example. A controlled language only helps in practice if it stays readable for engineers while still making the requirement more explicit [3]. The survey therefore complements the unseen-requirement review below, which carries the main support for the coverage claim. It does not evaluate whether engineers can independently author correct CNL translations from new requirements.

4.4.3 Evaluation

4.4.3.1 Validation Procedure

The unseen-review procedure followed the same semantic logic as the earlier CNL work, but it was now applied to material outside the design set. The unseen set contained 198 requirements from multiple ECUs and was selected to give rough spread across the derived class families while still preferring structurally difficult material. Each requirement was read manually and decomposed into atomic semantic statements when needed. A translatability label was then assigned, together with the relevant primitive label or labels when the meaning was covered by the frozen language. A requirement was counted as compound translatable only when every atomic statement in the decomposition could be represented by existing primitives.

The reviewed requirements were checked against the frozen primitives through manual decomposition and translation logic. The evaluation therefore tests actual CNL coverage on reviewed requirements.

4.4.3.2 Validation Labels

Five validation labels were used in the unseen review.

- **Fully translatable.** The entire requirement maps directly to one frozen primitive in one execution layer, without decomposition into multiple independent semantic statements. In other words, one primitive is sufficient to cover the whole meaning.

- **Compound translatable.** The entire requirement is still fully translatable, but only after decomposition into multiple atomic semantic statements. Each part maps to existing frozen primitives, so more than one primitive or more than one layer is needed even though no essential meaning is lost.
- **Partially translatable.** Part of the requirement is representable, but at least one essential semantic statement has no matching frozen primitive. The gap is therefore a missing semantic capability, not a composition issue.
- **Ambiguous.** The requirement wording is too under-specified, vague, or incomplete to determine a signal-level translation without changing the intended meaning.
- **Unsupported.** The requirement belongs to a semantic family or execution model that the three-layer CNL was not designed to represent, such as diagnostic capability declarations or broader computation specifications. This is a design boundary of the artifact.

The distinction between these labels was important for the interpretation of coverage. Fully translatable and compound translatable requirements were both counted as covered because the frozen CNL could represent the entire content. The difference was whether one primitive covered the full requirement directly or whether decomposition into several covered parts was necessary. Partially translatable requirements were not counted as covered because some essential meaning remained outside the primitive set. Ambiguous and unsupported requirements were also not counted as covered, but for different reasons. Ambiguous cases were blocked by source-quality problems, whereas unsupported cases lay outside the semantic families that the artifact was designed to represent. This separation is also consistent with ISO/IEC/IEEE 29148, which treats ambiguity, incompleteness, and poor verifiability as requirement-quality problems that should be exposed and not silently forced into controlled form [13].

4.4.3.3 Coverage Results

Table 4.4 summarizes the unseen validation outcome.

Together, the fully translatable and compound translatable categories give 166 covered requirements out of 198 reviewed requirements. This corresponds to a raw coverage of 83.8%. The unsupported cases were then excluded to show how well the frozen CNL handled the requirements that still belonged to its intended semantic target. After removing the 24 unsupported requirements, 174 in-scope requirements remained. Of these, 166 were covered, which gives an in-scope coverage of 95.4%. This result was measured on a non-random unseen sample and was scored by the authors during manual review. It therefore indicates artifact feasibility within the intended scope.

The raw 83.8% result is the more conservative number because it reports the full unseen set without exclusions. Taken together, the two numbers show that the CNL still has clear boundaries, but that inside those boundaries the frozen primitive set covers most unseen requirements.

Validation outcome	Count	% of 198	Interpretation
Fully translatable	140	70.7%	One frozen primitive covered the full requirement.
Compound translatable	26	13.1%	Multiple atomic statements were present, but all were covered by existing primitives.
Partially translatable	3	1.5%	Some content was covered, but at least one essential semantic statement was missing from the primitive set.
Ambiguous	5	2.5%	The source wording was too under-specified for translation without changing the intended meaning.
Unsupported	24	12.1%	The requirement belonged to a semantic family or execution model that the artifact did not support.

Table 4.4: Outcome of the unseen-requirement validation review in Cycle IV.

4.4.3.4 Failure Analysis

The partially translatable cases pointed to a narrow semantic gap around more complex storage logic, especially comparisons or selections between two stored sources. These cases were still within the general application domain, but they required capabilities beyond the current Storage / Lifecycle primitives.

Second, the ambiguous cases were caused by under-specified source wording and not by missing CNL expressiveness. Phrases such as “last known good value” or temporally bounded behavior without a concrete observable completion condition did not provide enough information for consistent CNL rewriting without meaning change. In these cases, the correct outcome of the validation was to block translation and not invent semantics. This follows the same requirement-quality logic discussed in ISO/IEC/IEEE 29148, where ambiguous or incomplete wording should be treated as a defect in the source requirement [13].

Third, the unsupported cases belonged to a few known excluded semantic areas. Some were service or policy declarations and not condition-action behavior. Others described broader stateful computation, arithmetic accumulation, or iterative multi-step handling that would require a different execution model from the three frozen layers. These cases confirm that the main boundary of the CNL is semantic, not just syntactic.

These results support keeping the primitive set compact. A CNL should add a primitive only when repeated evidence shows the same recurring meaning. Rare or highly specific cases should not force the language to support more behavior than most requirements need. The small set of uncovered in-scope cases instead shows

where future extensions may be needed without weakening the current artifact.

4.4.3.5 Cycle IV Main Result

Cycle IV showed that the frozen CNL v2 was usable and covered 166 of 198 unseen requirements, which is 83.8% raw coverage. When the 24 unsupported requirements are excluded, the in-scope coverage was 95.4% (166 of 174). The main result was that the remaining failures followed clear patterns. The uncovered cases fell into known semantic areas such as under-specified wording, diagnostic or policy artifacts, broader computation logic, and more complex storage policies. Cycle IV refinement also tightened the translation boundary for source lists whose conjunction semantics were not explicit.

This result shows that CNL v2 and the primitive set from Cycle III still worked on unseen requirements. The validation led to refinement and produced the final CNL v3 artifact. The result is clear enough to review with engineers, covers most in-scope unseen requirements, and shows where translation should stop.

4.5 Produced Artifact

Figure 4.21 summarizes the final artifact, from taxonomy families to the frozen primitives in each supported layer. A primitive was used for CNL design only when the same behavior appeared consistently across multiple examples. Cases with unclear, algorithmic, or unsupported behavior were kept outside the current grammar target.

4.5.1 Representative Translated Reference Dataset

After the class system stabilized, a representative translated reference dataset was created by selecting ten requirements per class where enough suitable examples existed. It included clear examples, edge cases, and structurally difficult cases so that grammar development would not be based only on simple formulations.

The dataset supports three design tasks. First, it gives concrete examples for defining class boundaries. Second, it provides example translations that guide CNL surface syntax. Third, it creates a stable basis for parser and grammar development. These translations also served as source material for the Cycle IV survey in Appendix G.

The full reference dataset is not included in the thesis appendix because it is derived from Volvo-internal requirement material. Representative examples from this dataset are instead shown throughout Chapters 4 and 5, including the survey examples discussed in Cycle IV.

4.5.2 CNL and Grammar Basis

The final software-level CNL v3 builds on the CNL v2 frozen in Cycle III and is designed to support the relevant class families from Cycle I and the stable primitives from Cycle III. Its purpose is to make the requirement structure clear enough for

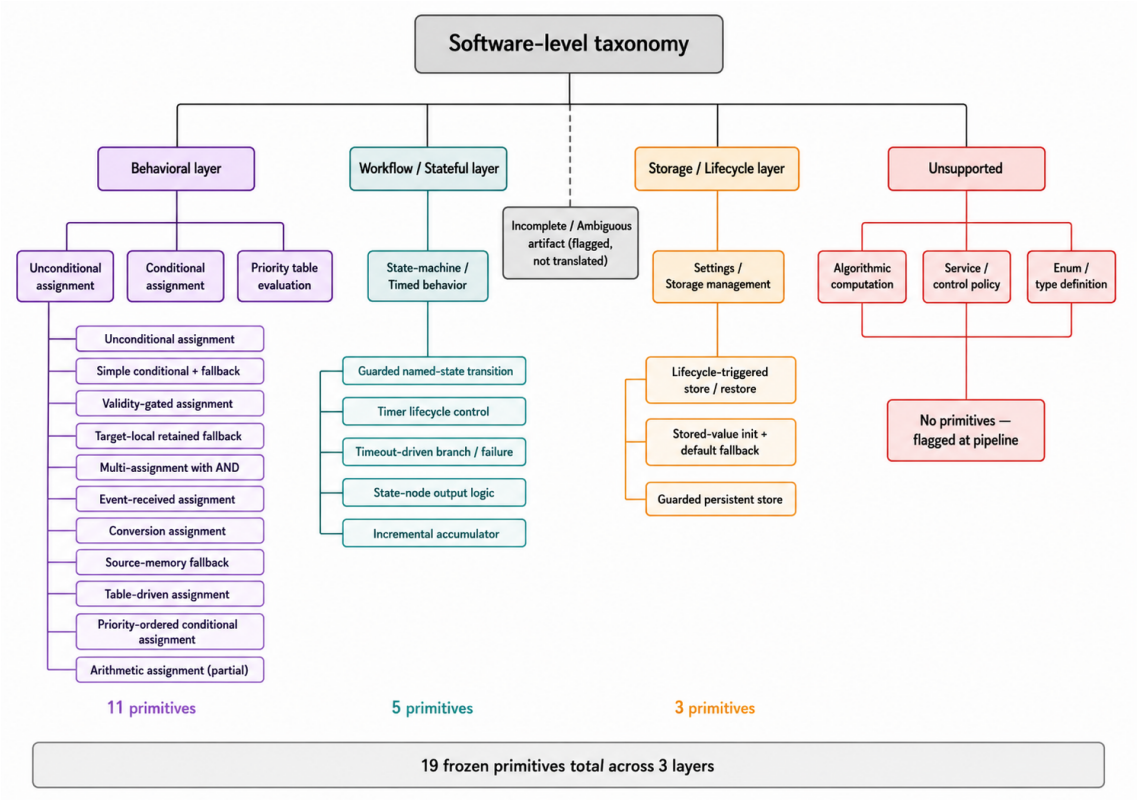


Figure 4.21: Overview of the software-level taxonomy families, their mapping to CNL layers, and the frozen primitives within each supported layer.

parsing and later validation while remaining readable for engineers. A compact codebook for the final frozen primitive set is provided in Appendix B. The grammar is specified in EBNF-style form and implemented with Lark [12]. The full Lark implementation of the grammar is provided in Appendix C. The full Behavioral, Workflow / Stateful, and Storage / Lifecycle CNL specifications are provided in Appendices D, E, and F. A conforming CNL statement can be parsed into a tree and transformed into an intermediate representation that exposes signals, operators, values, conditions, timing elements, and state relations.

The layered structure is important because the supported families do not all use the same execution semantics. Direct condition-action logic belongs to the Behavioral layer. State-dependent and transition-driven requirements belong to the Workflow / Stateful layer. Stored values, persistence, initialization, and lifecycle events belong to the Storage / Lifecycle layer. The grouped frozen primitive set is reported in Table 4.2.

The grammar does not force every observed requirement into the supported language. This boundary is part of the artifact. Requirements that are incomplete, ambiguous, algorithmic, highly policy-like, or dependent on unresolved metadata should be flagged instead of translated unsafely.

4.5.2.1 Worked Translation Example

This example shows how one requirement is traced through the artifact so that the codebook, grammar, and layer specification can be read together. In this thesis, these steps were applied manually by the authors. The example is illustrative and uses anonymized placeholder names so that the connection between the artifact parts is easier to see.

Step 1. Original requirement. The following requirement is used as a simple Storage / Lifecycle example:

> *"The value provided in Signal_A shall be stored in StoredSignal_A unless Signal_A is a missing signal, in which case the stored data in StoredSignal_A shall remain unchanged."*

Step 2. Classification. The requirement belongs to the *Guarded persistent store* primitive in Appendix B. In the codebook, this primitive is placed in the Storage / Lifecycle layer and is defined as follows. "A live value is written to explicit retained storage only when the store guard allows it." The canonical CNL form in the same entry is "When <WriteGuard>, then <LiveValue> shall be stored in <StoredDestination>. When <NonStoreCondition>, then <LiveValue> shall not be stored in <StoredDestination>."

Step 3. Layer specification. The full Storage / Lifecycle specification in Appendix F places this primitive in Section 5.3, *Guarded persistent store*. There the primitive is defined as persistence write behavior controlled by a guard. The same

section requires one live source value, one explicit stored destination, one explicit guard, and one explicit non-store case.

Step 4. Grammar basis. The controlled form is expressed as two ordinary `When` statements. In Appendix C, the outer statement shell is `when_statement`. The two effects are parsed by `stored_in_assignment` and `not_stored_assignment`. The non-store condition uses the shared `missing_predicate`. The write guard is written as the complement of that condition.

Step 5. Controlled form. Using the codebook and the layer specification together, the requirement becomes:

> `When not (Signal_A is missing), then Signal_A shall be stored in StoredSignal_A.`
>
> `When Signal_A is missing, then Signal_A shall not be stored in StoredSignal_A.`

The original requirement states the non-store case directly but leaves the write guard implicit. The controlled form makes both sides explicit so the result can be parsed and checked.

Source element	**Controlled-form element**
Live value	`Signal_A`
Stored destination	`StoredSignal_A`
Non-store condition	`Signal_A is missing`
"stored data in StoredSignal_A shall remain unchanged"	`Signal_A shall not be stored in StoredSignal_A`
Implicit write guard	`not (Signal_A is missing)`

Table 4.5: Element mapping for the guarded persistent-store worked example.

4.5.3 Pipeline

The pipeline design defines how a raw Requirements Engineering Tool artifact should be handled before CNL translation. Figure 4.22 summarizes the designed pipeline as an activity-style diagram.

The pipeline separates the main requirement statement from supporting text. In this thesis, *notes* are explicitly marked supporting text, commonly introduced by `Note:` or `Notes:`. *Context* denotes surrounding unmarked explanatory text that may help interpretation but is not necessarily part of the enforceable requirement. Both are preserved as supporting text, while only the isolated main requirement statement proceeds to classification and translation.

It then classifies the requirement according to the Cycle I class system and records whether it is textual, tabular, or mixed. It also detects translation-relevant flaws such

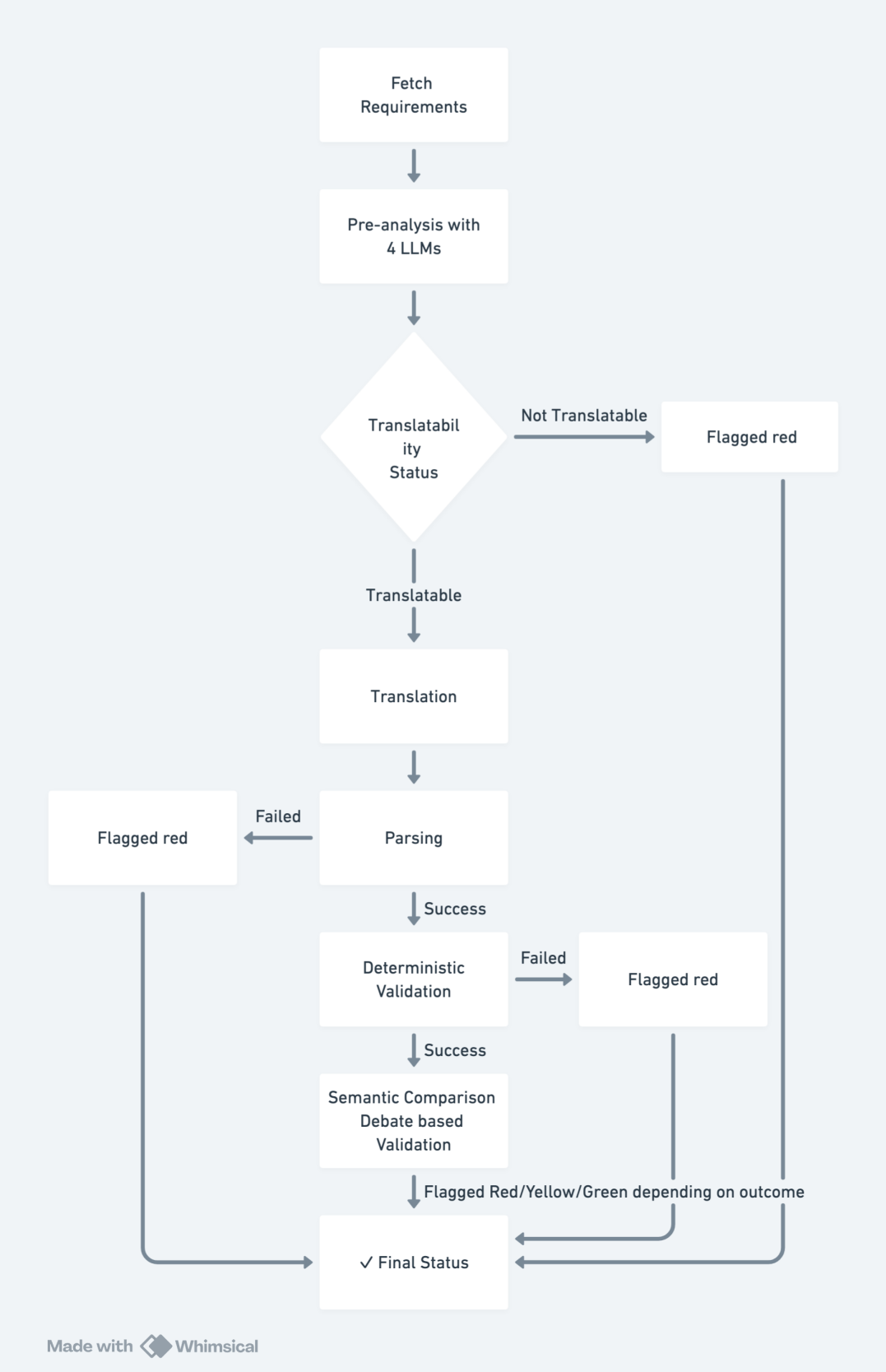


Figure 4.22: Designed pipeline for preparing requirements before CNL translation, parsing, validation, and semantic comparison.

as ambiguity, missing references, compound statements, unverifiable wording, and incomplete signal information. The possible translatability statuses are translatable, translatable with flags, or not safely translatable. Requirements judged not safely translatable should stop before CNL generation and return the identified issues for human review.

Accepted requirements are translated into the CNL, parsed, validated against available metadata for deterministic properties, and later assessed for semantic preservation between the original main requirement statement and the translated CNL. Semantic validation is treated as a separate evaluation concern because syntactic validity does not by itself prove that the translation preserved the intended meaning.

5

Discussion

5.1 Interpretation of the Results

The thesis followed a design science research approach across four cycles, and the results show why each cycle was necessary. Every cycle exposed a specific design problem that had to be solved before a stable CNL could be justified.

The main design lesson is that CNL development should start with empirical classification and semantic validation. A grammar built directly from a few example requirements would likely miss important classes or claim support for requirements that cannot be translated without changing the intended meaning or inventing missing information. The taxonomy gives the structural basis for grammar design. It defines which requirement classes the CNL must be able to express. Primitive discovery identifies the recurring atomic behaviors inside each class, for example unconditional assignment, guarded state transition, or guarded persistent store. Layer separation then assigns each primitive to the execution model it actually requires. These are Behavioral, Workflow / Stateful, and Storage / Lifecycle. This means the grammar only has to support behaviors it can represent without changing meaning.

Cycle I was necessary, but it could not answer the whole problem on its own. Broad classes stabilized and gave a usable map of the software-level requirements, yet the later analysis showed that one broad class could still contain several recurring semantic mechanisms. This matters because broad class stability could otherwise have created false confidence. A requirement that looked like ordinary conditional assignment at the class level could still contain retained fallback, storage behavior, table-driven logic, or workflow progression once it was examined more closely.

Cycle II introduced atomic semantic statements as the main unit of analysis and replaced broad-class saturation with semantic-pattern saturation. This shifted the thesis from broad structural classification toward the recurring meanings already present in the requirements. Atomic decomposition made it possible to see which semantic units actually recurred across datasets and which ones were only local wording variants or unsafe compounds. The resulting primitive set gave a clear basis for later grammar decisions.

The layered model is also a key contribution. Cycle III showed that the supported primitives did not share one execution interpretation even when they could be expressed in one controlled language. Direct condition-action behavior, workflow

or stateful progression, and storage or lifecycle behavior required different semantic treatment after parsing. This matters because a single, flat execution model would force condition-action, stateful, and storage requirements into the same interpretation. For example, treating a guarded persistent-store requirement as if it were an ordinary assignment would silently drop its persistence and lifecycle meaning. Separating the layers keeps these execution differences explicit. The pipeline design then uses this layered, classification-driven design as an ordered review path. A requirement is classified, its supporting text is separated, its translatability is checked, and only then is it translated and validated.

The Cycle IV validation results should also be interpreted carefully. The main point is not only that the frozen primitive set covered most unseen requirements inside the intended scope. The CNL was often judged clearer and less ambiguous than the original wording, but some verbose or table-heavy rewrites were still less preferred. This means semantic clarity and reader preference do not always increase together. The artifact therefore needs both semantic precision and readable presentation.

For the same reason, blocking ambiguous or unsupported requirements should be treated as a strength. The thesis does not aim to maximize apparent coverage by inventing missing semantics or by stretching the grammar into a broader programming notation. When the source wording is under-specified, ambiguous, policy-like, algorithmic, or dependent on unsupported semantic mechanisms, the safer result is to stop translation and preserve the reason for that stop. This makes the artifact more trustworthy as a basis for later industrial use.

5.2 Contribution

This thesis is closest to systematic, empirically grounded CNL construction such as Rimay [9], which derived a CNL for functional requirements in the financial domain and validated it on unseen specifications. Like that work, this thesis derives its language from real industrial requirement material, uses saturation to decide when the supported set is stable, and validates coverage on unseen requirements instead of using only constructed examples. Building a CNL from a dataset is already an established idea. The contribution of this thesis is four more specific points.

First, the study transfers this empirical, classification-driven approach to a setting that prior CNL work has not addressed. It focuses on software-level automotive requirements managed in a Requirements Engineering Tool, where requirement meaning often depends on surrounding metadata, signal references, and table structure in addition to the sentence itself. Structured approaches such as EARS and FRET [4], [5] and CNLs such as Rimay [9] target functional or behavioral requirements whose main meaning is mostly contained in the sentence. The software level studied here forced the taxonomy and grammar to handle supporting-text separation and metadata dependence explicitly.

Second, the resulting CNL is layered by execution semantics instead of expressed as one flat grammar. Behavioral, Workflow / Stateful, and Storage / Lifecycle requirements are separated because primitives that look similar on the surface can

require different execution interpretations after parsing. Existing CNL approaches for automotive requirements, such as the Structured English used by Bertram et al., organize requirements through temporal scopes and behavioral patterns and omit requirements that cannot be expressed in the language [10]. In this thesis, storage and lifecycle behavior is treated as a separate execution layer because its meaning depends on explicit retained-state and persistence semantics, such as reading from a stored value, writing to a stored value, initialization from a stored value with default fallback, or lifecycle-triggered store and restore. This is different from Workflow / Stateful behavior, whose meaning depends on process progression, transitions, timers, or timeouts.

Third, the artifact treats translatability as an explicit, reportable boundary. The pipeline does not try to maximize apparent coverage by forcing ambiguous, under-specified, or unsupported requirements into the language. This mirrors the boundary used by Bertram et al., who exclude requirements that are not expressible in their Structured English [10]. The evaluation also reports both raw coverage, 83.8%, and in-scope coverage, 95.4%, so the scope boundary remains visible. The contribution goes beyond stopping unsafe translations. It treats that boundary as a designed output of the artifact and quantifies it.

Fourth, the study refines how saturation is applied. Broad-class saturation alone produced a stable structural map but could create false confidence, because one class could still contain several distinct semantic mechanisms. Moving the stopping rule to semantic-pattern saturation at the atomic-statement level was what actually stabilized the primitive set.

For a reader outside Volvo, the transferable result is the method and these design lessons, not the Volvo-specific taxonomy. Another organization could apply the same classification-first, semantics-before-grammar, translatability-gated method to its own dataset while expecting a different concrete taxonomy.

Lessons learned

- Build the CNL from real requirement material. Generic templates such as EARS and FRET help, but they do not capture all domain-specific structures.
- Broad requirement classes are not enough on their own. Different semantic patterns can still exist inside the same class, so atomic decomposition is needed before grammar design.
- Separate requirements by execution meaning. Immediate behavior, process progression, and stored-value behavior should not be treated as one flat execution model.
- Do not force unclear requirements into a clean-looking translation. If the source is ambiguous or incomplete, stopping for human review is safer than inventing missing meaning.
- Report both raw coverage and in-scope coverage. This keeps the boundary of the language visible.

5.3 Answers to the Research Questions

RQ1 (Problem investigation). What software-level requirement classes can be derived from reviewed and translated industrial automotive requirements, and what structural properties describe them?

Answer: the reviewed software-level requirements form a stable nine-family taxonomy with clear, cross-ECU-stable class boundaries.

Cycle I showed that the reviewed software-level requirements can be organized into a stable nine-family taxonomy with clear class boundaries and cross-ECU stability. The main result was not only a list of classes. It was a grounded structural map of the material. The taxonomy separated recurring assignment-style requirements, more stateful or workflow-shaped requirements, storage-related requirements, and cases that were incomplete, ambiguous, or otherwise outside the intended translation scope. This gave the thesis a clear basis for later primitive discovery and CNL design.

This answer comes mainly from Cycle I. The class-discovery review and later cross-ECU checks reported in Chapter 4 showed that the taxonomy stabilized and held across the datasets. This supports the claim that the final taxonomy captures recurring structural properties in the software-level material and is not tied only to one local wording style inside ECU_1.

RQ2 (Artifact design). How can the derived classes and recurring primitive meanings be used to build a layered CNL and grammar basis, and how should a pipeline use this basis before translation and validation?

Answer: the classes become a layered CNL by moving from class labels to recurring semantic primitives, freezing the supported subset, and separating it into Behavioral, Workflow / Stateful, and Storage / Lifecycle layers that a pre-analysis pipeline design applies before translation and validation.

Cycle II and Cycle III showed how the broad classes could be turned into a layered CNL, a grammar basis, and a pipeline design for translation review. The key step was to move from class labels to recurring semantic units. Cycle II discovered primitive candidates from decomposed requirements, and Cycle III froze the supported subset, defined its translation boundaries, and separated the accepted semantics into Behavioral, Workflow / Stateful, and Storage / Lifecycle layers. This made it possible to connect controlled syntax to clear execution meaning instead of treating every supported requirement as if it behaved in the same way.

The same logic also shaped the pipeline design. It uses the class system and the frozen primitives to review requirements before later translation and validation. This answer therefore comes from Cycle II and Cycle III together. Cycle III formalized the frozen primitive set, the grammar basis, and the pipeline design presented in Chapter 4. Together, these cycles show how the empirical class system became a usable artifact for software-level requirements.

RQ3 (Evaluation). To what extent do the taxonomy, primitive set, layered CNL, and current validation steps support stable and usable CNL design,

and what limitations remain?

Answer: within the intended scope the frozen artifact covered 95.4% of unseen requirements in scope (83.8% raw), and practitioners generally judged the CNL clearer and less ambiguous, although some table-heavy rewrites were less preferred.

Cycle III and Cycle IV showed that the frozen primitive set and layered CNL support stable and usable CNL design within the intended scope, but they also made the remaining limits visible. Stability was supported first through the freeze review in Cycle III. Candidate primitives were only accepted when repeated examples showed one stable meaning, one consistent CNL form that did not change the intended meaning or invent missing information, and a clear execution-layer fit. This gave the language a tighter boundary and reduced the risk that one primitive would absorb several different mechanisms.

Cycle IV then tested usability and coverage in two different ways. The survey gave formative evidence about readability, ambiguity, and practitioner preference. It showed that the CNL was generally judged clearer and less ambiguous than the original wording, even though some table-heavy or verbose rewrites were less preferred. The unseen-requirement review gave the main empirical basis for the coverage claim and showed 83.8% raw coverage on unseen material. When the 24 unsupported requirements are excluded, the in-scope coverage was 95.4% (166 of 174). These results support the practical usefulness of the artifact, but they do not remove the remaining boundaries. The remaining limitations are discussed in Section 5.4.

5.4 Threats to Validity and Limitations

The taxonomy was derived through manual review and interpretation. Class assignment, class-boundary refinement, and treatment of ambiguous artifacts may therefore vary between researchers. This also applies to Cycle II and Cycle III. Atomic decomposition of compound requirements is partly interpretive, and different researchers might disagree about where one semantic statement ends and another begins. Primitive labeling is also partly interpretive because the analyst must decide whether an example belongs to an existing candidate, requires a new candidate, or should remain outside the supported set. Compound requirements therefore depend on reliable atomic statement extraction before translation, and incorrect decomposition could affect translation quality. These threats were reduced by iterative review, comparison between authors, use of an explicit incomplete or ambiguous category, selective translated cross-ECU checks, later classification across all ECU datasets in the material, and explicit freeze criteria in Cycle III.

The qualitative analysis is based on four interviews. Practitioner availability in the industrial setting made it difficult to schedule more one-on-one interviews. The participants were knowledgeable and represented different requirement-related roles, but the sample is still too small to claim that the themes represent the whole organization or all requirement-related roles. The unseen-requirement validation set and the survey also have limits. The unseen set contained 198 requirements that were

deliberately selected to include a rough spread across the derived class families while also preferring structurally difficult material. This made the validation useful, but the unseen set was not a statistically random sample from the full requirement set. A different selection strategy might have produced somewhat different raw and in-scope coverage values. The survey reflects a limited participant group. Eleven practitioners answered the survey, which is enough for a meaningful formative signal but not enough to support broad quantitative claims about organization-wide preference or adoption. The survey still supports the overall readability and usability of the CNL, while also showing that some table-heavy or verbose rewrites were not preferred. The survey participants also came from the same industrial context as the rest of the study, so their preferences may reflect local terminology, practices, and expectations. The survey evaluated readability and usability of selected translations, but it did not replace formal coverage validation or parser-level conformance testing.

The empirical material comes from one industrial setting and a limited set of ECU datasets. The cross-ECU classification and selective translation checks strengthen confidence inside the Volvo software-level context, but transfer to other organizations, domains, or requirement levels requires additional evaluation. This also applies to the grammar basis and primitive set. They are grounded in this material and have not yet been shown to transfer automatically to other companies, requirement cultures, or artifact repositories.

The current artifact is intentionally scoped. Some semantic areas remain outside the current CNL scope, including diagnostic service declarations, publication policies, complex timed processes, filtering and smoothing behavior, recurrence-based computation, and multi-step procedures. Broader algorithmic computation, service or control policy logic, enum or type definition artifacts, and more complex storage behaviors are also not fully covered by the frozen grammar. Some requirements were only partially translatable because they required primitives or operators that were not frozen in the first version, such as min or max operations or more complex memory policies. This scope boundary is part of the artifact, and it limits claims about universal translatability. The grammar basis and pipeline logic have also not yet been deployed as a full industrial workflow or tested through a full end-to-end automated trial. The pipeline guided the validation work in this thesis, but the study still does not show operational runtime performance or full tool integration. The thesis therefore supports the artifact as a grounded design basis. It is not yet a fully operational production system.

A further scope limitation is that the artifact targets only the software-requirement level. Functional requirements were not modeled in their own right. Logical requirements are expected to fall within reach because they are a semantic subset of software requirements, but they were not themselves part of the analyzed material and remain to be evaluated directly. The taxonomy, primitive set, and grammar therefore say little about how functional or logical requirements should be structured or translated, and extending the approach to those levels remains open.

5.5 Future Work

The most immediate next step is to implement and evaluate the full automated pipeline end to end. This should integrate classification, atomic decomposition, LLM translation, translatability gating, and semantic validation in one workflow. A stronger semantic-validation stage should also be implemented and evaluated so that meaning preservation can be checked more systematically. This should include the lightweight multi-agent debate-based structure discussed earlier in the thesis, where several LLM agents challenge competing translations before a final output is accepted.

Controlled scope extension is the next step on the CNL side. The current grammar boundary should stay fixed unless repeated examples justify extending it. New support should be added only when repeated examples show stable semantics and a clear execution interpretation. The most natural extensions include min and max operations, additional arithmetic primitives, more complex memory and storage behavior, more complex initialization policies, diagnostic and service declarations, protocol and request-response workflows, periodic timer processes, recurrence-based computation, broader algorithmic computation, and service or control policy requirements.

Automated atomic statement extraction should also be improved through the LLM pipeline so that decomposition errors have less effect on translation quality. After that, the artifact should be evaluated on a larger unseen test set to assess parser success rate and translation accuracy at scale. Engineer studies should also be repeated with a larger and more diverse participant group, including requirement authors from different teams and experience levels, to determine whether the current usability findings hold beyond the current industrial context.

Future work should also examine transfer beyond the current setting. One next step is to examine whether functional requirements can be improved through writing guidance and whether the same classification-driven design logic holds for functional requirements. The taxonomy, primitive set, and grammar should also be tested on requirement datasets from other organizations or automotive domains to assess how well the approach transfers beyond its current context.

6

Conclusion

This thesis addressed the problem that automotive requirements in the organization vary in structure, can be unclear, and often depend on context outside the sentence itself. This makes translation, parsing, and validation difficult when the goal is to preserve intended meaning. The work therefore focused on building a clear basis for software-level requirements so that supported requirements can be translated in a controlled way and unsupported cases can be identified early.

The main artifact is a classification-driven and layered CNL basis together with a pipeline design for reviewing requirements before translation and validation. The final artifact includes the software-level taxonomy, the frozen primitive set, and the CNL v3 grammar basis. This gives practitioners a safer basis for review before translation and later parsing and validation.

Across the thesis, the work moved from problem investigation and broad classification to semantic discovery, controlled grammar design, and final validation. The result is a grounded artifact that covers most unseen requirements that still fall inside the intended scope and that makes its own boundaries clear.

The artifact still has clear limits. It is intentionally scoped, grounded in one industrial setting, and not yet tested through a full end-to-end automated pipeline. Even so, the thesis shows that a useful CNL for automotive requirements should be built from empirical classification, stable primitive meanings, and clear scope boundaries. The current artifact gives a strong basis for later engineering and controlled extension.

Bibliography

[1] M. Staron, *Automotive Software Architectures: An Introduction*, 2nd ed. Springer, 2021. DOI: 10.1007/978-3-030-65939-4.

[2] K. M. Habibullah et al., "Requirements and software engineering for automotive perception systems: An interview study," *Requirements Engineering*, vol. 29, pp. 25–48, 2024. DOI: 10.1007/s00766-023-00410-1.

[3] A. Wyner et al., "On controlled natural languages: Properties and prospects," in *Controlled Natural Language*, Springer, 2010, pp. 281–289.

[4] A. Mavin and P. Wilkinson, "Ten years of EARS," *IEEE Software*, vol. 36, no. 5, pp. 10–14, Sep. 2019.

[5] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann, and N. Shi, "Formal requirements elicitation with FRET," in *Joint Proceedings of REFSQ 2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track*, 2020.

[6] I. Darif, G. El Boussaidi, S. Kpodjedo, and C. Politowski, "Controlled natural language for requirements specification: A systematic literature review," *ACM Computing Surveys*, 2025. DOI: 10.1145/3778169.

[7] V. Lucantonio, "Enhancing the consistency between requirements and test cases through the definition of a controlled natural language," M.S. thesis, Mälardalen University, May 2015.

[8] K. Mokos and P. Katsaros, "A survey on the formalisation of system requirements and their validation," *Array*, vol. 7, p. 100 030, 2020. DOI: 10.1016/j.array.2020.100030.

[9] A. Veizaga, M. Alferez, D. Torre, M. Sabetzadeh, and L. Briand, "On systematically building a controlled natural language for functional requirements," *Empirical Software Engineering*, vol. 26, no. 4, p. 79, 2021. DOI: 10.1007/s10664-021-09956-6.

[10] V. Bertram, H. Kausch, E. Kusmenko, H. Nqiri, B. Rumpe, and C. Venhoff, "Leveraging natural language processing for a consistency checking toolchain of automotive requirements," in *2023 IEEE 31st International Requirements Engineering Conference (RE)*, 2023, pp. 212–222. DOI: 10.1109/RE57278.2023.00029.

[11] L. Zhao et al., "Natural language processing for requirements engineering: A systematic mapping study," *ACM Computing Surveys*, vol. 54, no. 3, p. 55, 2021. DOI: 10.1145/3444689.

[12] E. Shinan, *Lark: A modern parsing library for python*, https://github.com/lark-parser/lark, 2023.

[13] *ISO/IEC/IEEE 29148:2018 systems and software engineering — life cycle processes — requirements engineering*, ISO/IEC/IEEE, 2018.

[14] Y. Du et al., "Context length alone hurts LLM performance despite perfect retrieval," *arXiv preprint arXiv:2510.05381*, 2025.

[15] P. K. Danso, M. S. Hasan, N. Balasubramanian, and O. Chowdhury, "Syntax is easy, semantics is hard: Evaluating LLMs for LTL translation," in *Proceedings of the 2026 ACM Secure Development Conference (SecDev '26)*, 2026. DOI: 10.1145/3805773.3806005.

[16] R. Bommasani et al., "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2022.

[17] P. Liang et al., "Holistic evaluation of language models," *Transactions on Machine Learning Research*, 2023.

[18] T. R. McIntosh et al., "Inadequacies of large language model benchmarks in the era of generative artificial intelligence," *arXiv preprint arXiv:2402.09880*, 2024.

[19] Center for AI Safety, Scale AI, and HLE Contributors Consortium, "A benchmark of expert-level academic questions to assess AI capabilities," *Nature*, vol. 649, pp. 1139–1147, 2026. DOI: 10.1038/s41586-025-09962-4.

[20] M. Oriol, Q. Motger, J. Marco, and X. Franch, "Multi-agent debate strategies to enhance requirements engineering with large language models," *arXiv preprint arXiv:2507.05981*, 2025.

[21] J. Chun, J. Li, I. Ahmed, and Q. Chen, "Is multi-agent debate (MAD) the silver bullet? an empirical analysis of MAD in code summarization and translation," *arXiv preprint arXiv:2503.12029*, 2025.

[22] Y. Liu et al., "Agent design pattern catalogue: A collection of architectural patterns for foundation model based agents," *arXiv preprint arXiv:2405.10467*, 2024.

[23] A. Hevner, S. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[24] E. Knauss, "Constructive master's thesis work in industry: Guidelines for applying design science research," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering Education and Training*, 2021.

[25] S. Jamshed, "Qualitative research method-interviewing and observation," *Journal of Basic and Clinical Pharmacy*, vol. 5, no. 4, pp. 87–88, 2014. DOI: 10.4103/0976-0105.141942.

[26] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[27] S. K. Ahmed et al., "Using thematic analysis in qualitative research," *Journal of Medicine, Surgery, and Public Health*, vol. 6, p. 100 198, 2025. DOI: 10.1016/j.glmedi.2025.100198.

[28] P. Mayring, "Qualitative content analysis," *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, vol. 1, no. 2, Art. 20, Jun. 2000.

[29] B. G. Glaser, "The constant comparative method of qualitative analysis," *Social Problems*, vol. 12, no. 4, pp. 436–445, 1965. DOI: 10.2307/798843.

[30] M. M. Hennink, B. N. Kaiser, and V. C. Marconi, "Code saturation versus meaning saturation: How many interviews are enough?" *Qualitative Health Research*, vol. 27, no. 4, pp. 591–608, 2017. DOI: 10.1177/1049732316665344.

A

Qualitative Analysis Material

A.1 Interview Guide

A.1.1 Introduction (5 min)

Before starting each interview, the authors introduced themselves, explained the study being performed at the company, and asked if the interview could be recorded.

A.1.2 Professional Experience (10 min)

1. Can you shortly introduce yourself and tell me which team you work with?
2. How long have you worked in the industry?
3. What is your current role in the company?
4. What does this role entail?

A.1.3 General Questions (45 min)

1. Have you worked with requirements for software projects? How do you interact with them? Do you write, review, or test them?
2. What type of requirement do you typically work with? Functional, logical, software design, or multiple types?
3. How are requirements typically written in your team today, for example free text, templates, or tables?
4. Are there any internal conventions or guidelines when writing or reviewing requirements? Do you follow them? If there are any, is it mandatory to follow them?
5. When writing or reviewing a requirement, what information do you assume the reader already knows?
6. What are the most common issues you encounter with requirements in practice, for example ambiguity, missing context, or inconsistent structure?
7. How often do you need additional context, for example signals, documents, or other artifacts, to fully understand a requirement?

8. Can you give an example of a requirement that you think is well written? What makes it good?
9. Can you give an example of a requirement that is difficult to understand or problematic? Why?
10. How do you handle requirements that are incomplete or unclear? Do you have to talk to the person who wrote the requirement?
11. Do you think the use of a controlled natural language (CNL) would be beneficial in your work? Why or why not?
12. Is there anything you would like to add? Anything we have missed, or something else you want to add to the interview?

A.2 Current Codebook

Table A.1: Current consolidated codebook after analysis of Interviews I1–I4.

Code	Definition	How this shaped the artifact
inconsistent_writing_standard	Use when requirement writing is described as lacking a consistently enforced standard, or when authors express a clear need for one shared structure.	Supports the need for one standardized target CNL and conformance checking.
author_dependent_requirement	Use when requirements are said to vary depending on who wrote them, which team wrote them, or how different practitioners respond to standardization efforts.	Supports the need for translation into a controlled form and structured writing rules.
abstraction_level_affects_clarity	Use when clarity, required detail, or interpretability is described as varying across architectural, functional, logical, or software-level requirements.	Supports abstraction-aware scoping, translation patterns, and validation rules.
explicit_signal_references_support_validation	Use when explicit signal references are described as improving clarity or being necessary for deterministic validation.	Supports signal-aware CNL translation and validation.
missing_information_in_tool	Use when the interviewee says required information is missing in the Requirements Engineering Tool or linked artifacts.	Pipeline must support missing-field detection and flagging.
traceability_incomplete	Use when trace links are said to be partial, missing, or unreliable.	Pipeline should support traceability-aware validation and context quality checks.
validation_depends_on_external_metadata	Use when validation is said to depend on linked artifacts, trace context, glossaries, databases, signal catalogs, or other information outside the requirement sentence itself.	Validation should retrieve and use external context sources where needed.

Code	Definition	How this shaped the artifact
supporting_context_may_help	Use when surrounding context, comments, or notes are described as potentially helpful for understanding or locating meaning.	The artifact should preserve supporting context instead of discarding it blindly.
supporting_context_should_be_separate	Use when comments, notes, or surrounding context are described as redundant, misplaced, or better kept outside the core requirement statement.	The artifact should separate the core CNL sentence from optional supporting context.
table_representation_may_help	Use when the interviewee sees tables as useful for clarity, mappings, or repeated structures.	Table-like representations may be appropriate for some requirement classes.
table_rep_can_be_confusing	Use when tables are described as hard to interpret or semantically unclear.	Table types need classification before translation.
expert_interpretation_needed	Use when the interviewee implies that some cases cannot be resolved without expert knowledge.	Human review remains necessary in difficult or underspecified cases.
audience_dependent_interpretation	Use when interpretation, amount of explanation, or usefulness of context is said to depend on the intended audience or reader role.	The artifact should distinguish core machine-checkable logic from optional explanatory information for different readers.
formal_language_for_validation_exists	Use when the interviewee describes an existing formalized language, grammar, or compiler-based approach for validating requirement logic.	Supports the feasibility of machine-checkable CNL structures and compiler-assisted validation.
guideline_driven_requirement_evolution	Use when explicit guideline documents or newer documented practices are said to shape newer requirements differently from older legacy artifacts.	The pipeline may need to distinguish between guideline-shaped newer material and older legacy requirements.
non_standardizable_requirement_exception	Use when a requirement is described as too exceptional, complex, or unusual to safely force into one standardized structure.	The pipeline must support explicit exception handling and avoid unsafe translation.

Code	Definition	How this shaped the artifact
logical_completeness_required	Use when validation is said to depend on checking whether all relevant signal values, states, or table combinations are covered.	The pipeline should support logical completeness checks in addition to surface-level format validation.

B

Frozen Primitive Codebook

This appendix provides a compact codebook for the final frozen primitive set used in the thesis artifact. It summarizes the 19 primitives included in the final layered CNL basis. The primitives listed below correspond to the three supported layers shown in Figure 4.21. Full normative definitions, exclusions, and translation rules remain in Appendices D, E, and F.

Table B.1: Compact codebook for the final frozen primitive set.

Primitive	**Layer**	**Explanation**	**Canonical CNL Form**
Unconditional assignment	Behavioral	Immediate assignment with no guarding condition.	<Target> shall be set to <Value>.
Simple conditional assignment and explicit fallback	Behavioral	Current-state condition triggers one immediate assignment, with an explicit fallback branch.	When <Condition>, then <Target> shall be set to <Value>. Otherwise, <Target> shall be set to <FallbackValue>.
Validity-gated assignment	Behavioral	Assignment is allowed only when explicit validity or range predicates are satisfied.	When <Source> is within Valid Signal Range, then <Target> shall be set to <Source>. Otherwise, <Target> shall be set to <FallbackValue>.
Target-local retained fallback	Behavioral	The target keeps its own last assigned or last valid value as an explicit fallback.	Otherwise, <Target> shall keep its last valid value.
Multi-assignment with AND	Behavioral	One condition governs several coordinated immediate actions.	When <Condition>, then <TargetA> shall be set to <ValueA> AND <TargetB> shall be set to <ValueB>.
Event-received immediate assignment	Behavioral	Receipt of an explicit event directly triggers one or more immediate deterministic assignments.	When <ReceivedEvent> is received, then <Target> shall be set to <Value>.
Conversion assignment	Behavioral	A source value is assigned after applying a named conversion reference.	When <Source> is within Valid Signal Range, then <Target> shall be set to <Source> converted using <ConversionTable>.
Source-memory fallback	Behavioral	An invalid or missing source is replaced by that source's last valid value for one immediate assignment or substitution.	When <Source> is within Invalid Signal Range, then <Target> shall be set to last valid value of <Source>.

Primitive	Layer	Explanation	Canonical CNL Form
Table-driven assignment	Behavioral	An output value is selected from a tabular mapping from one or more input conditions.	<Target> shall be set according to the following table: <Input> = <Case1> → <Result1>; Otherwise → <FallbackValue>.
Priority-ordered conditional assignment	Behavioral	Overlapping prose conditions are evaluated top-to-bottom with first-match-wins priority.	(Priority 1): When <ConditionA>, then <Target> shall be set to <ValueA>. Otherwise, <Target> shall be set to <FallbackValue>.
Arithmetic assignment (partial)	Behavioral	Only the direct current-input arithmetic subset is frozen; arithmetic that depends on remembered state remains outside scope.	<Target> shall be set to <Source> / <Divisor>.
Guarded named-state transition	Workflow / Stateful	A state machine transitions from one named state to another when explicit guards are satisfied.	When in state <SourceState> AND <Condition>, then state shall transition to <TargetState>.
Timer lifecycle control	Workflow / Stateful	A named timer or counter is initialized, started, stopped, or reset under an explicit guard.	When <Condition>, then <Timer> shall start counting from <StartValue>.
Timeout-driven branch / failure transition	Workflow / Stateful	Timeout expiry itself is the decisive trigger for a branch, state change, or status assignment.	When <Message> has not been received within <Timeout> ms, then <Status> shall be set to <Value>.
State-node output logic	Workflow / Stateful	Assignments or output logic execute while the system remains inside a named state node.	When in state <StateName>, then <Output> shall be set to <Value>.
Incremental accumulator	Workflow / Stateful	A numeric target updates itself by adding or subtracting a delta, optionally at a repeated interval.	When <Condition>, then <Accumulator> shall be incremented by <Delta> every <Period>.

Primitive	Layer	Explanation	Canonical CNL Form
Lifecycle-triggered store / restore	Storage / Lifecycle	A live value is read from or written to retained storage at an explicit lifecycle boundary.	When <LifecycleEntry>, then <Target> shall be read from <StoredSource>. When <LifecycleExit>, then <Target> shall be written to <StoredDestination>.
Stored-value initialization with default fallback	Storage / Lifecycle	Initialization reads from explicit stored state and uses a default only when the stored value is invalid or inaccessible.	<Target> shall be read from <StoredSource>. If <StoredSource> cannot be accessed or contains an invalid value, then <Target> shall be set to <DefaultValue>.
Guarded persistent store	Storage / Lifecycle	A live value is written to explicit retained storage only when the store guard allows it.	When <WriteGuard>, then <LiveValue> shall be stored in <StoredDestination>. When <NonStoreCondition>, then <LiveValue> shall not be stored in <StoredDestination>.

C

CNL Grammar Implementation

This appendix provides the full Lark implementation of the final software-level CNL grammar used in the thesis artifact. It gives one parseable grammar file that covers the Behavioral, Workflow / Stateful, and Storage / Lifecycle layers together. The normative layer-specific specifications are provided separately in Appendices D, E, and F. Together, the specifications and the grammar form the full CNL artifact. The specifications explain the intended design of each layer, while this appendix shows how those rules were encoded in the parser. Section references such as "§4.3" or "§5.2" inside the grammar comments refer to sections within those layer-specific specification documents, not to sections of the thesis chapter. The listing is included as an implementation artifact rather than as a new analytical result.

```
// CNL Grammar for Automotive Requirements
//
// This file is a unified layered grammar covering:
// - Layer 1: Frozen Behavioral CNL (behavioral_cnl_specification.md §4, §9)
// - Layer 2: Backward-compatibility / provisional (parseable, not frozen)
// - Layer 3: Frozen Workflow / Stateful CNL (workflow_stateful_cnl_specification.md §5, §10)
// - Layer 4: Frozen Storage / Lifecycle CNL (storage_lifecycle_cnl_specification.md §5, §8)
// The section references in the comments below point to those specification
// documents, not to sections of the main thesis text.
//
// Design:
// - One unified grammar because all layers share conditions, identifiers,
//   values, operators, and the When/then shell.
// - Layer-specific forms add only minimal action phrases.
// - Parsing is layer-agnostic. Semantic classification into behavioral /
//   workflow / storage is a downstream pass.
//
// Canonical frozen-v1 output prefers:
// - no leading "The" in unconditional assignments
// - a comma after "Otherwise"
// - trailing periods

start: statement+

statement: unconditional_statement
         | when_statement
         | priority_when_statement
         | otherwise_statement
         | initialization_statement
         | table_statement
         | guarded_table_statement
         | state_transition_statement
         | state_initial_transition_statement
         | state_exit_statement
         | state_node_statement

// ============================================================================
// LAYER 1 - FROZEN BEHAVIORAL CNL STATEMENT SHELL (§4.1-§4.11)
```

```
// ============================================================================

// Unconditional statement (§4.1).
// Optional leading "The" retained for backward compatibility only.

unconditional_statement: "The"? signal_list "shall" unconditional_action "."?

unconditional_action: set_unconditional_action
                    | send_unconditional_action
                    | converted_unconditional_action

set_unconditional_action: "be"? "set" "to" value ("for" duration)?
send_unconditional_action: NOT_KW? "be"? "sent"
converted_unconditional_action: "be"? "converted" "using" identifier

// Initialization (§4.2)
initialization_statement: "At" "initiation" target "shall" "be"? "set" "to" value "."?

signal_list: target ("," target)* ","?

// ============================================================================
// CONDITIONAL / OTHERWISE (§4.3, §4.4, §4.5, §4.11)
// ============================================================================

// Standard When/then (§4.3)
when_statement: "When" condition "," "then"? assignment_list "."?

// Priority-ordered When/then (§4.11)
priority_when_statement: "(" "Priority" NUMBER ")" ":" "When" condition "," "then"? assignment_list
    "."?

// Otherwise fallback (§4.4)
otherwise_statement: "Otherwise" ","? assignment_list "."?

// ============================================================================
// LAYER 1 - TABLE-DRIVEN ASSIGNMENT (§4.10)
// ============================================================================

table_statement: target_list "shall" "be" "set" "according" "to" "the" "following" "table" ":"
    table_body (otherwise_statement)?

guarded_table_statement: "When" condition "," "then"? target_list "shall" "be" "set" "according" "to
    " "the" "following" "table" (("AND"|"and") assignment)* ":" table_body (otherwise_statement)?

target_list: target (("and"|",") target)*

table_body: table_header table_separator table_row+
table_header: "|" column_name ("|" column_name)* "|"
table_separator: "|" TABLE_SEP ("|" TABLE_SEP)* "|"
table_row: "|" cell_value ("|" cell_value)* "|"

TABLE_SEP: /---+/

column_name: identifier
           | "Priority"

cell_value: value
          | range_cell
          | comparator_cell
          | compound_cell
          | passthrough_cell
          | dont_care
          | catch_all
          | priority_number
          | keep_last_cell
          | keep_last_valid_cell

keep_last_cell: "keep" "its" "last" "value"
keep_last_valid_cell: "keep" "its" "last" "valid" "value"
range_cell: value "-" value
```

```
106 comparator_cell: (OP|EQUAL) value
107 compound_cell: comparator_cell (("AND"|"and") comparator_cell)+
108 passthrough_cell: "=" identifier
109 priority_number: NUMBER
110 dont_care: DONT_CARE
111 catch_all: CATCH_ALL_OTHERWISE_SIGNAL_MISSING | CATCH_ALL_SIGNAL_MISSING | catch_all_otherwise
112
113 catch_all_otherwise: "Otherwise"
114
115 DONT_CARE: "-"
116 CATCH_ALL_OTHERWISE_SIGNAL_MISSING.3: "Otherwise or signal missing"
117 CATCH_ALL_SIGNAL_MISSING.2: "Signal missing"
118
119 // ============================================================================
120 // LAYER 1 - FROZEN BEHAVIORAL ASSIGNMENTS (§4.1-§4.9)
121 // ============================================================================
122
123 assignment_list: assignment (("AND"|"and") assignment)*
124
125 assignment: frozen_assignment
126           | workflow_assignment
127           | storage_assignment
128           | compatibility_assignment
129
130 frozen_assignment: set_assignment
131                  | keep_last_value_assignment
132                  | keep_last_valid_value_assignment
133                  | source_memory_assignment
134                  | treated_as_last_valid_assignment
135                  | bounded_duration_assignment
136                  | send_assignment
137                  | send_for_at_least_assignment
138
139 set_assignment: target ("shall" "be"? "set" "to" | EQUAL) value
140 keep_last_value_assignment: target "shall" "keep" "its" "last" "value"
141 keep_last_valid_value_assignment: target "shall" "keep" "its" "last" "valid" "value"
142 source_memory_assignment: target ("shall" "be"? "set" "to" | EQUAL) "last" "valid" "value" "of"
        identifier
143 treated_as_last_valid_assignment: target "shall" "be" "treated" "as" "its" "last" "valid" "value"
144 bounded_duration_assignment: target "shall" "be"? "set" "to" value "for" duration
145 send_assignment: target "shall" NOT_KW? "be"? "sent"
146 send_for_at_least_assignment: target "shall" "be"? "sent" "for" "at" "least" duration
147
148 // ============================================================================
149 // LAYER 2 - COMPATIBILITY / PROVISIONAL ASSIGNMENTS
150 // ============================================================================
151
152 compatibility_assignment: compatibility_keep_value_assignment
153
154 compatibility_keep_value_assignment: target "shall" "keep" "its" "previous" "value" ("for" duration)
        ?
155                                    | target "shall" "keep" "its" "last" "value" "for" duration
156
157 // ============================================================================
158 // LAYER 3 - FROZEN WORKFLOW / STATEFUL CNL (§5.1-§5.5)
159 // ============================================================================
160
161 // --- §5.1 Guarded named-state transition ---
162
163 // Node-based: When in state X AND <guards>, then state shall transition to Y.
164 state_transition_statement: "When" "in" "state" identifier ("AND"|"and") condition
        temporal_qualifier? "," "then"? "state" "shall" "transition" "to" identifier "."?
165                          | "When" identifier EQUAL identifier ("AND"|"and") condition
        temporal_qualifier? "," "then"? identifier "shall" "be"? "set" "to" identifier "."?
166
167 // State shall initially transition to X.
168 state_initial_transition_statement: "State" "shall" "initially" "transition" "to" identifier "."?
169
170 // Exit: When <trigger>, then <machine> shall exit.
171 state_exit_statement: "When" condition "," "then"? identifier "shall" "exit" "."?
```

```
172
173 // --- §5.4 State-node output logic ---
174
175 // When in state X [AND guards], then <assignments>.
176 // When entering state X [AND guards], then <assignments>.
177 state_node_statement: "When" "in" "state" identifier (("AND"|"and") condition)? "," "then"?
        assignment_list "."?
178                     | "When" "entering" "state" identifier (("AND"|"and") condition)? "," "then"?
        assignment_list "."?
179
180 // --- §5.2 Timer lifecycle control ---
181 // --- §5.5 Incremental accumulator ---
182 // These appear as assignments within When/then statements.
183
184 workflow_assignment: timer_lifecycle_assignment
185                    | accumulator_assignment
186
187 // Timer lifecycle (§5.2)
188 timer_lifecycle_assignment: target "shall" "be"? "initialized" "to" value
189                           | target "shall" "start" "counting" "from" value
190                           | target "shall" "stop" "counting"
191                           | target "shall" "be"? "reset"
192
193 // Incremental accumulator (§5.5)
194 accumulator_assignment: target "shall" "be"? "incremented" "by" value ("every" period)?
195                       | target "shall" "be"? "decremented" "by" value ("every" period)?
196
197 period: NUMBER DURATION_UNIT
198       | DURATION_UNIT
199       | identifier DURATION_UNIT
200       | "cycle" "time"
201
202 // ============================================================================
203 // LAYER 4 - FROZEN STORAGE / LIFECYCLE CNL (§5.1-§5.3)
204 // ============================================================================
205
206 // Storage-specific action phrases:
207 // - shall be read from <StoredSource>
208 // - shall be written to <StoredDestination>
209 // - shall be stored in <StoredDestination>
210 // - shall not be stored in <StoredDestination>
211
212 storage_assignment: read_from_assignment
213                   | written_to_assignment
214                   | stored_in_assignment
215                   | not_stored_assignment
216
217 read_from_assignment: target "shall" "be"? "read" "from" identifier
218 written_to_assignment: target "shall" "be"? "written" "to" identifier
219 stored_in_assignment: target "shall" "be"? "stored" "in" identifier
220 not_stored_assignment: target "shall" "not" "be" "stored" "in" identifier
221
222 // ============================================================================
223 // FROZEN CONDITIONS (§9, shared by all layers)
224 // ============================================================================
225
226 condition: or_condition temporal_qualifier?
227 or_condition: and_condition (("OR"|"or") and_condition)*
228 and_condition: not_condition (("AND"|"and") not_condition)*
229 not_condition: ("NOT"|"not")? comparison
230              | ("NOT"|"not")? "(" condition ")"
231
232 comparison: identifier (OP|EQUAL) value
233           | identifier
234           | transition
235           | range_predicate
236           | received_predicate
237           | missing_predicate
238           | not_received_within_predicate
239           | abs_comparison
```

```
          | has_not_passed
          | has_expired
          | last_valid_value_comparison
          | no_valid_value_predicate

transition: identifier "transitions" "from" value "to" value
range_predicate: identifier "is" NOT_KW? "within" RANGE_KIND "Signal" "Range"
received_predicate: received_subject "is" "received"
missing_predicate: identifier "is" "missing"
not_received_within_predicate: received_subject "has" "not" "been" "received" "within" value
     DURATION_UNIT? ("of" identifier)?
abs_comparison: "abs" "(" identifier ")" (OP|EQUAL) value
has_not_passed: identifier "has" "not" "yet" "passed"
has_expired: identifier "has" "expired"
last_valid_value_comparison: "last" "valid" "value" "of" identifier (OP|EQUAL) value
no_valid_value_predicate: "no" "valid" "value" "of" identifier "exists"

temporal_qualifier: "for" "more" "than" value DURATION_UNIT?
                  | "for" "the" "duration" "of" value

// Shared duration surface
duration: NUMBER DURATION_UNIT       -> duration_with_unit
        | value                      -> duration_value

NOT_KW: "not" | "NOT"
RANGE_KIND: "Valid" | "Invalid"

EQUAL: "="
OP: "≥" | "≤" | ">" | "<" | "≠"

// ============================================================================
// IDENTIFIERS AND VALUES (shared by all layers)
// ============================================================================

bus_qualifier: "(" "on" CNAME ")"

?reference: CNAME bus_qualifier? -> plain_reference
          | CNAME "of" reference -> subsignal_reference

received_subject: RECEIVED_SUBJECT

target: reference
identifier: reference

named_value: reference
value: expr | QUOTED

?expr: term ((PLUS|MINUS|SLASH|STAR) term)*
?term: atom
?atom: multi_word_name
     | named_value
     | NUMBER
     | "(" expr ")"

multi_word_name: CNAME CNAME+

PLUS: "+"
MINUS: "-"
SLASH: "/"
STAR: "*"

// Combined name -- supports dotted names like Signal.SubField
CNAME: /[A-Za-z_][A-Za-z0-9_]*(\.[A-Za-z_][A-Za-z0-9_]*)*/

// Context-specific multi-word subject for the frozen received predicate.
// Intentionally does not broaden general references or identifiers.
RECEIVED_SUBJECT: /(?:(?!\b(?:OR|AND|or|and)\b)(?!\s+is\s+received\b)(?!\s+has\s+not\s+been\s+
     received\b)[^\n])+?(?=\s+(?:is\s+received|has\s+not\s+been\s+received)\b)/

// ============================================================================
```

```
308 // TERMINALS
309 // ============================================================================
310
311 // Numbers: integers and decimals
312 NUMBER: /[+-]?[0-9]+(\.[0-9]+)?/
313
314 // Duration units shared by all layers
315 DURATION_UNIT.3: "ms" | "s" | "min" | "second" | "seconds" | "minute" | "minutes"
316
317 // Quoted strings
318 QUOTED: /"[^"]*"/ | /'[^']*'/
319
320 // ============================================================================
321 // WHITESPACE AND COMMENTS
322 // ============================================================================
323
324 %import common.WS
325 %ignore WS
326
327 COMMENT: /\/\/[^\n]*/
328 %ignore COMMENT
329
330 NEWLINE: /\n/
331 %ignore NEWLINE
```

D

Behavioral CNL Specification

This appendix provides the full normative Behavioral CNL specification used as part of the thesis artifact. The specification records the layer boundary, frozen primitives, and translation rules that support the grammar basis discussed in Chapter 4 and keeps the artifact self-contained inside the thesis.

````
# Controlled Natural Language (CNL) Specification

This document is the normative design specification for Behavioral CNL in `vbc-cnl`.

This specification is organized around the final accepted primitives and a clear scope boundary:

* the final primitive review decides what is included
* scope is determined by meaning, not by source class labels
* the grammar is limited to validated Behavioral CNL constructs
* candidate extensions, non-behavioral families, and deferred patterns are documented only to make the boundary explicit

The live grammar in [grammar.lark](src/vbc_cnl/compiler/grammar/grammar.lark) should align to the frozen constructs in this document. Sections that describe candidate extensions, out-of-scope families, or deferred patterns are not grammar-expansion requests.

---

## 1. Purpose

Behavioral CNL is a narrow requirement language used as an intermediate representation between free-form source requirements and downstream deterministic checks.

Its core job is to represent immediate behavioral rules of the form:

```text
When <condition>, then <immediate assignment(s)>.
```

The language is deliberately conservative. It is not intended to absorb every recurring source shape. It exists to capture the validated stateless behavioral core and to make the boundary to adjacent semantic families explicit.

### 1.1 Behavioral scope boundary

Behavioral CNL is defined by four constraints:

* stateless evaluation
* immediate effect
* explicit condition -> action structure
* deterministic semantics

Behavior is in scope only when the operative rule can be reduced to current-state evaluation and immediate effect without hidden execution machinery.

The following are not part of Behavioral CNL:

````

```
* source-memory semantics beyond the narrow frozen subset in §4.9 -- including reset-reference state, stored prior values of another signal, non-validity-triggered ‘last known value‘, and source-memory embedded in workflow, timer, or storage contexts
* timer or counter processes that evolve over time
* workflow / protocol sequencing, request-response matching, retry, abort, handshake, or wait semantics
* storage / persisted-value semantics
* stateful computations such as filtering, moving averages, accumulation, or iterative conversion
* hidden policy tables or lifecycle assumptions that are essential to the result

---

## 2. High-level source classification (non-normative)

Source classes remain useful as descriptive intake buckets, but they are not used to decide scope or grammar promotion. A row tagged as ‘algorithmic computation‘ may still contain a narrow in-scope arithmetic subset, while a row tagged as ‘unconditional assignment‘ may still be out of scope if it depends on hidden remembered state.

| Source class | Typical use | Scope role |
| --- | --- | --- |
| Unconditional assignment | direct assignments, passthrough, subsignal assignment | descriptive only |
| Conditional assignment | ‘When‘ / ‘Otherwise‘ rules, range gating, mapping tables | descriptive only |
| Priority table evaluation | ordered first-match ladders | descriptive only |
| State-machine / timed behavior | transitions, timers, counters, timed actions | descriptive only |
| Algorithmic computation | formulas, ratios, filtering, conversion, accumulation | descriptive only |
| Settings / storage management | persistence, stored values, initialization from storage | descriptive only |
| Interface / control policy | external interfaces, workflows, publication policies | descriptive only |
| Enum / type definition | metadata artifacts | descriptive only |
| Incomplete / ambiguous artifact | structurally unsafe source material | descriptive only |

The normative sections below are organized by primitive behavior and validated boundary, not by these class labels.

---

## 3. Semantic classification framework

### 3.1 Scope status

| Scope status | Meaning |
| --- | --- |
| ‘in Behavioral CNL‘ | Fully inside the frozen behavioral core. The rule is stateless, immediate, deterministic, and reducible to explicit condition-action semantics. |
| ‘candidate behavioral extension‘ | The behavior is still atomic and behavioral, but one narrow semantic surface is not yet frozen. No hidden memory, workflow, or time evolution is required. |
| ‘future behavioral extension‘ | The behavior remains behavioral in spirit, but needs a new validated conceptual surface or still contains unresolved semantic structure. |
| ‘out of Behavioral CNL‘ | The requirement depends on a different execution model: source-memory, timer/process evolution, workflow/protocol semantics, lifecycle semantics, or multi-step/stateful computation. |

### 3.2 Translatability

Scope and translatability are independent.

| Translatability | Meaning |
| --- | --- |
| ‘full‘ | The full requirement can be expressed without semantic loss in the validated CNL surface. |
| ‘partial‘ | Only part of the requirement is expressible, or essential semantics remain outside the frozen surface. |
| ‘not safely translatable‘ | Normalization would materially change or invent semantics. |
```

````
Examples of independence:

* a row may be ‘out of Behavioral CNL‘ and still be partially translatable
* a row may be ‘candidate behavioral extension‘ and still have ‘partial‘ translatability

### 3.3 Common issue types

The validation workflow uses issue types to describe why a row is easy, boundary, or out of scope.

| Issue type | Meaning |
| --- | --- |
| ‘normalization‘ | Surface cleanup only: operators, table expansion, phrasing, punctuation, enum normalization, or explicit guard expansion |
| ‘ambiguous guards‘ | The rule boundary or precedence is not stable enough to freeze safely |
| ‘computation model‘ | A deterministic transform exists, but the transform surface is semantically essential and not yet frozen |
| ‘timer lifecycle + temporal evolution‘ | The row depends on a timer, counter, sample loop, or ongoing temporal process |
| ‘source-memory semantics‘ | The row depends on remembered source state, reset references, stored prior values, or non-target-local memory |
| ‘workflow / protocol‘ | The row depends on sequencing, interactions, retry/confirm flows, or interface protocol behavior |
| ‘lifecycle semantics‘ | The row depends on system/component lifecycle rather than an explicit named current-state condition |
| ‘state-machine declaration‘ | The row defines structural containers or lifecycle shells rather than one executable behavioral rule |

### 3.4 Governing rules

The following rules control all classifications in this specification:

* classify from operative semantics, not wrapper prose
* scope is determined by primitive behavior, not by workbook class labels
* primitive validation results are authoritative
* do not infer hidden state, hidden tables, or hidden policy just to make a row fit the language
* do not collapse stateful behavior into stateless condition-action rules

---

## 4. Behavioral CNL - Frozen primitives

This section lists the validated behavioral core. These primitives are frozen because the semantics are stable, repeatedly observed, and already fit the condition-action model without hidden state or process evolution.

### 4.1 Unconditional assignment

**Description**

One or more targets receive immediate values with no guarding condition.

**Allowed forms**

‘‘‘text
Signal_A shall be set to Value.
Signal_B shall be set to Value1.
Signal_C shall be set to Value2.
‘‘‘

**Examples**

* direct passthrough - ‘Signal_A shall be set to Signal_B.‘
* bus-qualified passthrough - ‘Field_A (on Bus_A) shall be set to Field_B (on Bus_B).‘
* subsignal assignment using ‘X of Y‘

**Normalization notes**

* omit articles before identifiers in canonical output
* preserve bus qualifiers as ‘(on BusName)‘
* preserve subsignal references as ‘X of Y‘ unless the source already uses dotted notation
````

````
* keep trailing periods in canonical output

### 4.2 Simple conditional assignment and explicit fallback

**Description**

Current-state conditions trigger immediate assignments, with optional explicit fallback.

**Allowed forms**

```text
When Condition, then Signal_B shall be set to Value.
Otherwise, Signal_B shall be set to Value.
```

**Examples**

* ‘When Signal_A = Signal_B, then Signal_A shall be set to Active. Otherwise, Signal_A shall be set to Inactive.‘
* ‘When Field_A = Upper, then Signal_A shall be set to Active. Otherwise, Signal_A shall be set to Inactive.‘

**Normalization notes**

* normalize ‘If‘ to ‘When‘
* normalize ‘ELSE‘ to ‘Otherwise‘
* prefer ‘Otherwise,‘ with a comma in new output
* use ‘Otherwise‘ when the second branch is the true complement of the first

### 4.3 Validity-gated assignment and valid-signal-range checks

**Description**

Assignment is gated by explicit signal-validity predicates or valid/invalid range checks.

**Allowed forms**

```text
When Signal_A is within Valid Signal Range, then Signal_C shall be set to Source.
When Signal_A is within Invalid Signal Range, then Signal_C shall be set to NotAvailable.
```

**Examples**

* valid/invalid range gating for wheel-speed derivation with explicit fallback
* signed-range pass-through with explicit out-of-range fallback

**Normalization notes**

* normalize source variants such as ‘is Valid signal‘ or ‘is a Valid signal‘ to ‘is within Valid Signal Range‘
* distribute shared range predicates across explicit signal names when needed
* keep quality/fallback semantics explicit rather than implicit

### 4.4 Target-local retained fallback

**Description**

The target may retain its own previously assigned value as an explicit fallback. Behavioral CNL freezes two target-local retention variants: plain retained fallback and validity-qualified retained fallback.

**Allowed forms**

```text
Otherwise, Signal_A shall keep its last value.
When Condition, then Signal_A shall keep its last value.
Otherwise, Signal_A shall keep its last valid value.
When Condition, then Signal_A shall keep its last valid value.
```
````

````
**Examples**

* valid-range pass-through with ‘Otherwise, Signal_A shall keep its last value.‘
* ‘When Timer_A > 0, then Signal_B shall keep its last value.‘
* ‘When Signal_C = Invalid Signal, then Signal_D shall keep its last valid value.‘
* ‘When Signal_E ≥ Threshold_A OR Signal_F ≥ Threshold_B, then Signal_G shall keep its last valid value.‘

**Meaning**

* ‘last value‘ means the target’s own most recent assigned value
* ‘last valid value‘ means the target’s own most recent valid assigned value
* both forms are target-local retention only
* ‘keep its last valid value‘ is a frozen Behavioral assignment variant separate from plain ‘keep its last value‘

**Accepted normalization examples for validity-qualified retained fallback**

```text
Target shall keep previous valid value.
Target shall retain its previous valid value.
Target shall remain at its previous valid value.
```

Normalize to:

```text
Target shall keep its last valid value.
```

**Normalization notes**

* this primitive is limited to target-local retention only
* ‘last value‘ here means the target’s own last assigned value
* ‘last valid value‘ here means the target’s own last valid assigned value
* do not silently normalize plain ‘keep its last value‘ into ‘keep its last valid value‘, or the reverse
* it is not source-memory fallback
* it does not mean the last valid value of another signal
* it does not cover ‘Signal_A shall be set to last received value of Source‘ or ‘Signal_A shall be set to last valid value of Source‘
* it does not cover ‘last value of Signal_A = Y‘ or ‘previous value of Signal_A = Y‘ used as a condition
* it does not cover ‘If no previous value is known, then Signal_B shall be set to Signal_A‘
* it does not cover timed holds, delays, until-regimes, or timeout-expiry behavior
* it does not cover hysteresis, accumulators, moving averages, delta-from-previous, highest-since-reset, or previous-run-cycle computations
* it does not imply PersistentStore_A, persistent storage, or Memory_A / volatile buffering
* it does not cover persistent-store save, restore, or initialization from stored state
* retain the fallback explicitly; do not encode it as hidden persistence behavior
* this form may also appear as a cell value (‘keep its last value‘) in table-driven assignment (§ 4.10) output columns, or as an ‘Otherwise‘ clause after a table body

### 4.5 Multi-assignment with ‘AND‘

**Description**

One condition governs multiple coordinated immediate actions.

**Allowed forms**

```text
When Condition, then Signal_A shall be set to X AND Signal_B shall be set to Y.
```

**Examples**

* one trigger sets three Signal_A-related fields under one shared condition
* one trigger sets a request active and applies a minimum-send-duration action
````

````
280
281 **Normalization notes**
282
283 * the condition is written once and governs the full assignment list
284 * keep the list flat; do not invent implicit sequencing between the actions
285
286 ### 4.6 Event-received immediate assignment
287
288 **Description**
289
290 Event-received immediate assignment captures rows where receipt of one explicit event, status,
      external notification, or message condition directly triggers one or more immediate
      deterministic assignments.
291
292 This primitive is Behavioral only when the received condition acts as an observable trigger for
      immediate assignment and does not require request/response correlation, timeout monitoring,
      first-receipt state, source-memory, storage, or workflow/protocol progression.
293
294 **Allowed forms**
295
296 ```text
297 When Event_A is received, then Signal_B shall be set to Value.
298 When Event_B is received OR Signal_A is received, then Signal_B shall be set to Value.
299 When Event_A is received, then Signal_E shall be set to Value1 AND Signal_F shall be set to Value2.
300 ```
301
302 **Structured-field rule**
303
304 If the source assigns fields of a structured target, the fields must be expanded as separate `X of Y
      ` assignments joined with `AND`.
305
306 Allowed:
307
308 ```text
309 When Event_A is received, then Field_A of Signal_A shall be set to Value_A AND Field_B of Signal_A
      shall be set to Value_B.
310 ```
311
312 Not allowed:
313
314 ```text
315 When Event_A is received, then Signal_A shall be set to Field_A = Value_A AND Field_B = Value_B.
316 ```
317
318 Reason:
319 Current CNL supports structured-field assignment plus coordinated `AND`-joined actions, but not
      inline record/object literal assignment.
320
321 **Examples**
322
323 * `When Event_A is received, then Field_A of Signal_A shall be set to Value_A AND Field_B of
      Signal_A shall be set to Value_B.`
324 * `When Event_B is received, then Field_A of Signal_A shall be set to Value_A AND Field_B of
      Signal_A shall be set to Value_C.`
325 * `When Event_C is received, then Field_A of Signal_A shall be set to Value_A AND Field_B of
      Signal_A shall be set to Value_D.`
326
327 **Normalization notes**
328
329 * preserve the received event/status/interface/message name exactly as far as possible
330 * if multiple received events are listed with `OR`, `either`, or `any`, normalize them as `OR`-
      joined received predicates
331 * the condition is written once and governs the full assignment list
332 * the assignment side must use existing Behavioral assignment forms
333 * structured target fields must use `Field of <Target>`
334 * do not use inline record/object literal values on the right-hand side
335
336 **Exclusions**
337
338 This primitive does not cover:
````

```
* request/response correlation
* timeout or missing-message monitoring
* first-receipt or first-valid gating
* until-received regimes
* retry, abort, send-until, or protocol progression
* latest/last/previously received value semantics
* source-memory or remembered prior-value semantics
* storage or persistence behavior
* generic Memory_A/volatile buffering
* received wording used only in descriptions, notes, wrappers, or table headers

**Boundary note**

‘received‘ remains a surface family, not a single semantic primitive. Rows containing ‘received‘
    must still be classified by their main meaning. Only the narrow event-received immediate
    assignment pattern above is frozen as Behavioral CNL.

### 4.7 Supporting frozen surfaces

These are not separate family names, but they are part of the frozen behavioral surface. Bounded
    duration assignment means ‘Signal_A shall be set to Value for Duration‘; it does not include
    timed retention such as ‘Target shall keep its last value for Duration‘, which remains outside
    the frozen v1 surface unless separately validated.

**Frozen condition surfaces**

* comparison predicates: ‘=‘, ‘≠‘, ‘>‘, ‘<‘, ‘≥‘, ‘≤‘
* boolean composition: ‘AND‘, ‘OR‘, ‘NOT‘, parentheses
* explicit transitions: ‘Signal transitions from A to B‘
* range predicates: ‘Signal is within Valid Signal Range‘, ‘Signal is within Invalid Signal Range‘
* narrow received predicate: ‘Signal_A is received‘ only for the frozen event-received immediate
    assignment primitive
* ‘abs()‘ conditions
* ‘has not yet passed‘
* source-memory condition: ‘last valid value of Source‘ with comparator (§4.9d)
* source-memory existence predicate: ‘no valid value of Source exists‘ (§4.9c)

**Frozen timing surfaces**

* ‘for more than Duration‘
* ‘for the duration of Duration‘
* bounded duration assignment: ‘Signal_A shall be set to Value for Duration.‘

**Examples**

* explicit transition trigger
* ‘for more than Duration‘
* ‘for the duration of Duration‘
* ‘has not yet passed‘
* bounded duration assignment surface

**Boundary note**

Timer values may appear as explicit current-state conditions, but timer lifecycle actions, counters,
     and clock-like processes are not frozen behavioral primitives.

### 4.8 Conversion assignment

**Description**

A target is assigned the value of a source signal after applying a named conversion. The conversion
    reference must be a named entity that exists in the Requirements Engineering Tool (typically a
    Local Constant or named calibration table). The CNL does not express the table contents or the
    interpolation method.

A standalone conversion declaration states that a signal uses a named conversion without specifying
    a target or gate. This form is behavioral when it captures a real conversion relationship and
    the referenced conversion exists.
```

````
**Allowed forms**

```text
Signal_A shall be converted using Table_A.
Signal_B shall be set to Source converted using Table_A.
When Condition, then Signal_C shall be set to Source converted using Table_A.
```

**Examples**

* `Signal_A shall be converted using Table_A.`
* `Signal_A shall be converted using Table_A.`
* `When Signal_A is within Valid Signal Range, then Signal_B shall be set to Signal_A converted using Table_A. When Signal_A = Error, then Signal_B shall be set to Error.`
* `When Signal_D is within Valid Signal Range, then Signal_A shall be set to Signal_D converted using Table_B. When Signal_D is within Invalid Signal Range, then Signal_A shall be set to NotAvailable.`

**Normalization notes**

* normalize `linearized according to` to `converted using`
* normalize `set to the linearized X` to `set to X converted using Table_A` where the conversion reference is resolved from the linked conversion-definition requirement
* the conversion table name must exist in the Requirements Engineering Tool; do not invent names
* inline calibration tables without a named Requirements Engineering Tool reference cannot use this form
* do not express table contents, interpolation method, or formula details in the CNL

**Boundary**

* inline 2-point calibration tables without a named Requirements Engineering Tool reference remain partial (only the error/invalid branch is translatable)
* piecewise interval formulas and explicit multi-point interpolation with inline formula remain candidate extensions (see §6.1)
* overflow guards that reference a conversion table's range limit remain candidate extensions (see §6.1)

### 4.9 Source-memory fallback

**Description**

Source-memory fallback captures the narrow pattern where a source signal's last valid value is used as a substitute or assignment when that source signal is currently invalid, missing, or in error. This is source-memory semantics -- it refers to the retained state of another signal, not to the target's own last assigned value.

This primitive is frozen only for the narrow immediate-assignment subset where:

* the retention trigger is an explicit validity/error condition on the source signal
* the effect is one immediate assignment or substitution
* source, target, and guard are all explicit
* no timer, workflow, storage persistence, or process evolution is required

**Allowed forms**

```text
When Signal_A is within Invalid Signal Range, then Signal_C shall be set to last valid value of Source.
When Signal_A = missing, then Signal_D shall be treated as its last valid value.
When no valid value of Source exists, then Signal_C shall be set to Signal_A.
```

**Canonical form variants**

§4.9a -- Assignment from source's last valid value:

```text
When Condition, then Signal_B shall be set to last valid value of Source.
```
````

````
§4.9b -- Source substitution (source treated as its own last valid value):

```text
When Condition, then Signal_B shall be treated as its last valid value.
```

§4.9c -- Initialization guard (no previous valid value exists):

```text
When no valid value of Source exists, then Signal_B shall be set to Signal_A.
```

**Examples**

* ‘When Field_A = missing, then Field_Signal_A shall be treated as its last valid value. When no valid value of Field_A exists, then Field_Signal_A shall be treated as Inactive.‘
* same pattern for a related signal
* ‘When Signal_A is within Invalid Signal Range, then Signal_A shall be treated as its last valid value.‘
* ‘When Signal_A is not within Valid Signal Range, then Signal_B shall be treated as its last valid value.‘
* ‘When Field_B is in error, then Field_Signal_A shall be treated as its last valid value.‘
* ‘When Signal_B = Invalid Signal, then Signal_C shall keep its last valid value. When no valid value of Signal_B exists, then Signal_A shall be set to Status_A.‘ (combines §4.4 target-local retention with §4.9c initialization guard)
* ‘When last valid value of State_A = Open, then ...‘ (source-memory used as a condition guard -- see §4.9d below)
* initialization with explicit default -- ‘When no valid value of State_B exists, then Field_A of Signal_D shall be set to Value_A.‘

**§4.9d -- Previous-value condition guard**

Source-memory may also appear as a condition rather than as an assignment target:

```text
When last valid value of Signal_A = Value, then Signal_C shall be set to Result.
```

This form is frozen only when:

* the remembered-value condition is explicit
* it controls one immediate assignment
* no timer, workflow, or storage persistence is required

**Normalization rules**

* ‘last known value of Source‘ normalizes to ‘last valid value of Source‘ **only** when the retention trigger is an explicit validity/error condition on the source signal (e.g., ‘Signal is within Invalid Signal Range‘, ‘Signal = missing‘, ‘Signal is not valid‘, ‘Signal is in error ‘)
* do NOT normalize ‘last known value‘  ‘last valid value‘ when the surrounding context suggests: last received value (even stale), last value before lifecycle exit, last stored value, last value before timeout, or other non-validity-triggered semantics. In such cases, preserve the original wording and route to the appropriate layer.
* ‘consider Source to have its last valid value‘ normalizes to ‘Signal_A shall be treated as its last valid value‘
* ‘shall be considered to have their respective last known value‘ normalizes to ‘shall be treated as its last valid value‘ (one per signal, when validity-triggered)
* ‘If no previous value is known, presume X‘ normalizes to ‘When no valid value of Source exists, then Signal_B shall be set to X.‘
* ‘If Source has no valid values‘ normalizes to ‘When no valid value of Source exists‘

**Meaning**

* ‘last valid value of Source‘ means the most recent valid value of a *different* signal (the source), not the target’s own last assigned value
* ‘shall be treated as its last valid value‘ means downstream logic shall use the source’s retained valid value as if it were the current value
* ‘When no valid value of Source exists‘ means no valid value has ever been observed for that source since initialization
````

````
* these forms are source-memory semantics, not target-local retention
* §4.4 (‘Target shall keep its last valid value‘) remains separate and covers only target-local retention

**Boundary**

* this primitive does NOT cover timed holds such as ‘keep last value for 4 seconds‘ (Workflow/Stateful)
* this primitive does NOT cover storage-governed fallback such as ‘last valid value before lifecycle exit‘ or ‘value stored in PersistentStore_A‘ (Storage/Lifecycle)
* this primitive does NOT cover ‘last received value‘ when the retention is not validity-triggered (route by context)
* this primitive does NOT cover source-memory inside a state-machine, timer lifecycle, or workflow progression context
* ‘last known value‘ that is NOT triggered by explicit validity/error condition must NOT be automatically normalized to ‘last valid value‘ -- it must be preserved and routed by semantic context

### 4.10 Table-driven assignment

**Description**

An output signal is assigned a value determined by a mapping table. The table maps one or more input conditions (column values) to an output value.

This primitive is the normative surface for requirements that contain tabular mappings. Two evaluation modes exist:

* **Non-priority tables** (default): rows are independent, conditions must not overlap. If source conditions overlap without explicit priority, exclusion guards are added during normalization.
* **Priority tables** (source explicitly states priority): rows are evaluated top-to-bottom, first matching row wins. A ‘Priority‘ column (1...n) is added during normalization.

**Canonical template -- standalone table**

‘‘‘text
<output> shall be set according to the following table:

| <input_1> | <input_2> | <output> |
| --- | --- | --- |
| value_a | value_b | result_1 |
| value_c | - | result_2 |
| Otherwise | Otherwise | fallback |
‘‘‘

**Canonical template -- guarded table (pre-condition + table)**

‘‘‘text
When <guard_condition>,
then <output> shall be set according to the following table:

| <input_1> | <output> |
| --- | --- |
| value_a | result_1 |
| value_b | result_2 |
| Otherwise | fallback |
‘‘‘

**Canonical template -- guarded table with side-effect**

‘‘‘text
When <guard_condition>,
then <output> shall be set according to the following table
AND <side_target> shall be set to <side_value>:

| <input_1> | <output> |
| --- | --- |
| value_a | result_1 |
| Otherwise | fallback |
‘‘‘
````

````
565
566 The ‘AND‘ side-effects use standard multi-assignment syntax (§4.5). The colon (‘:‘) terminates the
        statement header and introduces the table body.
567
568 **Canonical template -- multi-output table**
569
570 ‘‘‘text
571 <output_1> and <output_2> shall be set according to the following table:
572
573 | <input_1> | <input_2> | <output_1> | <output_2> |
574 | --- | --- | --- | --- |
575 | value_a | - | result_1a | result_1b |
576 | value_b | value_c | result_2a | result_2b |
577 | Otherwise | Otherwise | fallback_a | fallback_b |
578 ‘‘‘
579
580 When the intro names multiple outputs, the table has multiple output columns -- one for each named
        signal.
581
582 **Cell value conventions**
583
584 | Cell content | Meaning |
585 | --- | --- |
586 | Specific value or range (‘Value_A‘, ‘State_A‘, ‘Range_A‘, ‘> Threshold_A‘) | Match that value,
        enum, or range. Dash-ranges are inclusive on both endpoints (‘Range_A‘ means ‘≥ LowerBound_A
        AND ≤ UpperBound_A‘) |
587 | Compound condition (‘> Threshold_B AND ≤ Threshold_A‘) | Match when all sub-conditions are
        satisfied (used for exclusion guards) |
588 | ‘= <signal>‘ | Passthrough -- output equals input signal |
589 | ‘-‘ | Don’t care -- this input column is irrelevant for this row |
590 | ‘Otherwise‘ (in input column) | Catch-all -- matches any present value not matched by other rows.
        Must be the last row |
591 | ‘Signal missing‘ (in input column) | The input signal has not been received (availability fallback
        ). Placed above ‘Otherwise‘ |
592 | ‘Otherwise or signal missing‘ (in input column) | Combined catch-all: matches any unmatched
        present value OR signal not received. Must be the last row |
593
594 **Evaluation semantics**
595
596 There are two evaluation modes, determined by whether the source explicitly declares priority:
597
598 *Non-priority tables (default):*
599
600 * rows are independent -- conditions across rows MUST NOT overlap
601 * a row matches when ALL input column conditions are satisfied simultaneously
602 * ‘-‘ (don’t care) always satisfies the condition for that column
603 * ‘Otherwise‘ in the catch-all row matches any present value not matched by any other row
604 * ‘Signal missing‘ matches when the input signal has not been received -- it is a specific named
        condition, not a catch-all, and is placed above ‘Otherwise‘
605 * ‘Otherwise or signal missing‘ combines both: matches any unmatched present value OR signal not
        received
606 * row order is not semantically binding in non-priority tables. During normalization, the catch-all
        row (‘Otherwise‘ or ‘Otherwise or signal missing‘) is always moved to the last position
607 * if the source has overlapping conditions but does NOT explicitly state priority, the table must be
         disambiguated by adding exclusion guards (‘AND‘) to the overlapping cells
608 * if no row matches and no fallback row exists, the output is undefined (this is a spec error in the
         source) -- unless the source explicitly states a retain-default (see **Retained-value fallback
        ** below)
609
610 *Priority tables (explicitly marked):*
611
612 * rows are evaluated top-to-bottom -- first matching row wins
613 * the table includes a ‘Priority‘ column with values 1, 2, 3, ... n (top to bottom)
614 * a row matches when ALL input column conditions (excluding the Priority column) are satisfied
        simultaneously
615 * overlapping conditions are permitted -- row order resolves ambiguity
616 * a priority table is identified when the source explicitly uses keywords such as: "priority from
        top to bottom", "first match", "priority", "evaluated in order", or equivalent wording
617 * descriptive priority preambles (e.g., "Shall be set according to table with priority from top to
        bottom") are consumed into the Priority column and omitted from the CNL output
````

````
618
619 **Catch-all normalization**
620
621 Source catch-all and fallback wordings are normalized to one of three CNL keywords:
622
623 | Source wording | Normalized to |
624 | --- | --- |
625 | "Any other value", "All other values", "All other condition", "All other cases", "For all other
        conditions and values", "For all other conditions", "All other strings", "Default", "Otherwise",
         empty cell in what is clearly a catch-all row | `Otherwise` |
626 | "Signal missing" (standalone, when not a datatype value of the conditional signal) | `Signal
        missing` |
627 | "Any other value or signal missing", "Other value or signal missing", "Any other value or missing
        ", "All other values or signal missing", "All other values and combinations or signal missing",
         "Any other values or input signal(s) missing", "Any other combination of values or with signal
         missing" | `Otherwise or signal missing` |
628
629 **Normalization rules:**
630
631 * `Otherwise` -- pure value fallback. Matches any present-but-unmatched value. Must always be the **
        last row** in the table. In non-priority tables, if the source has the catch-all row in a non-
        last position, it is moved to the last row during normalization.
632 * `Signal missing` -- availability fallback. Means "signal not received". This is a specific named
        condition (like `Error` or `Not Available`) and can appear at any position. It is placed **
        above** `Otherwise`.
633 * `Otherwise or signal missing` -- combined fallback. Must always be the **last row**.
634
635 **"missing" disambiguation:** When the source uses the word "missing" in a table condition, check
        whether "Missing" is a datatype value of the conditional signal. If it is, treat it as a value
        match (a normal row). If it is not a datatype value, it means "signal not received" and is
        written as `Signal missing`.
636
637 **Priority table exception:** In priority tables, only the absolute last row (lowest priority) is
        normalized to `Otherwise`, `Signal missing`, or `Otherwise or signal missing` -- and only if
        that row is genuinely a catch-all. Intermediate broad conditions (e.g., "Any other value" at
        priority 2 when priority 3 still exists below) keep their original wording as-is, because row
        order is semantically binding.
638
639 **Same-output rows:** When multiple rows produce the same output (e.g., `Not available (7)` and `
        Signal missing` both map to the same result), they are always kept as separate rows. Merging is
         never performed.
640
641 **Retained-value fallback (no-match default)**
642
643 When the source explicitly states that the output shall keep its previous value if no row matches (e.
        g., "shall keep the previous value if there is no match in the table"), this is encoded as an `
        Otherwise` statement after the table body:
644
645 ```text
646 <output> shall be set according to the following table:
647
648 | <input_1> | <output> |
649 | --- | --- |
650 | value_a | result_1 |
651 | value_b | result_2 |
652
653 Otherwise, <output> shall keep its last value.
654 ```
655
656 Alternatively, when `keep its last value` applies to only one specific input condition (not as the
        table-level default), it appears as a cell value in the output column of that row:
657
658 ```text
659 | <input> | <output> |
660 | --- | --- |
661 | 0 | Inactive |
662 | 1 | keep its last value |
663 | Otherwise | Not Available |
664 ```
665
````

````
666 The distinction: if retain is the **table-level no-match default**, use `Otherwise` after the table.
        If retain is the **output for a specific input condition**, it is a cell value in that row's
        output column.
667
668 **Priority table form**
669
670 When the source explicitly declares priority ordering, the table includes a `Priority` column as the
        first column. The values are integers 1 through n, assigned top-to-bottom.
671
672 ```text
673 <output> shall be set according to the following table:
674
675 | Priority | <input_1> | <input_2> | <output> |
676 | --- | --- | --- | --- |
677 | 1 | value_a | value_b | result_1 |
678 | 2 | value_c | - | result_2 |
679 | 3 | Otherwise | Otherwise | fallback |
680 ```
681
682 The `Priority` column is added by normalization -- it is NOT an input condition column. Its sole
        purpose is to encode that row order is semantically binding (first-match-wins).
683
684 **Overlapping conditions in non-priority tables**
685
686 When the source table has overlapping row conditions but does NOT explicitly state priority, the
        overlapping cells must be disambiguated with exclusion guards:
687
688 Source (overlapping, no priority statement):
689 ```text
690 | Signal | Output |
691 | > Threshold_A | State_A |
692 | > Threshold_B | State_B |
693 ```
694
695 Normalized (exclusion guard added):
696 ```text
697 | Signal | Output |
698 | > Threshold_A | State_A |
699 | > Threshold_B AND ≤ Threshold_A | State_B |
700 ```
701
702 This preserves the source intent without introducing implicit priority semantics.
703
704 **Column structure rules**
705
706 * output column(s) are identified by the signal name(s) in the intro statement (`<output> shall be
        set according to...`) -- they are typically the rightmost column(s) but position is not
        normative
707 * when the intro names multiple outputs (e.g., "Signal_A and Signal_B shall be set..."), the table
        has multiple output columns
708 * all other columns are input condition columns (except `Priority`)
709 * if present, the `Priority` column is always the first column and is NOT an input condition -- it
        encodes evaluation order
710 * the header row defines signal/parameter names for each column
711 * separator row (`| --- | --- |`) is required between header and data
712 * no artificial column limit -- tables may have as many input columns as the requirement demands
713 * column count must be consistent across all rows
714
715 **Relationship to When/then syntax**
716
717 Table-driven assignment is the normative surface for tabular requirements. A table with N data rows
        is semantically equivalent to N `When` statements plus one `Otherwise` statement -- provided
        the table has no overlapping conditions or has already been disambiguated.
718
719 The table surface is required when:
720
721 * the source already contains a table (regardless of row count)
722 * the mapping is naturally a lookup or priority ladder
723
724 The table surface is preferred (but not required) when:
````

````
* there are 3+ distinct input-to-output rows that would otherwise expand to 3+ When/Otherwise statements

Single-condition or two-condition requirements without tabular source remain as When/Otherwise statements.

**Examples**

Simple 2-column table:

```text
Signal_A shall be set according to the following table:

| Signal_B | Signal_A |
| --- | --- |
| Value_A | State_A |
| Range_A | State_B |
| Otherwise or signal missing | Status_A |
```

Multi-input 3-column table:

```text
Signal_A shall be set according to the following table:

| Signal_B | Signal_C | Signal_A |
| --- | --- | --- |
| State_C | State_C | Result_A |
| State_D | - | Result_B |
| - | State_D | Result_B |
| Status_B | Status_B | Status_B |
| Otherwise or signal missing | Otherwise or signal missing | Status_C |
```

Guarded table with side-effect (non-priority, exclusion guards added for overlap):

```text
When Signal_A is within Invalid Signal Range,
then Status_A shall be set according to the following table
AND Signal_A shall be set to Status_B:

| Signal_A | Status_A |
| --- | --- |
| > Threshold_A | Result_C |
| < Threshold_B | Result_D |
| > Threshold_C AND ≤ Threshold_A | Result_E |
| < Threshold_D AND ≥ Threshold_B | Result_F |
| Otherwise | Result_G |
```

Priority table (source explicitly states "priority from top to bottom"):

```text
Signal_A shall be set according to the following table:

| Priority | Signal_B | Field_A | Field_B | Signal_A |
| --- | --- | --- | --- | --- |
| 1 | State_A | - | - | State_A |
| 2 | State_B | - | - | State_B |
| 3 | - | Value_A | active | State_A |
| 4 | - | Value_B | active | State_B |
| 5 | Otherwise or signal missing | - | Otherwise or signal missing | Signal_C |
```

**Normalization notes**

* the intro line always uses the canonical form: `<output> shall be set according to the following table:`
* source intro variants are normalized to the canonical form -- examples of source wordings that all
````

```
     normalize:
  - "shall be set according to table"  canonical form
  - "shall be according the following table"  canonical form
  - "shall provide the following mapping for X"  `Signal_A shall be set according to the following table:`
  - "shall be forwarded as"  identify output from column headers, produce canonical form
  - "see below table" / "see the table below"  identify output from column headers or surrounding text, produce canonical form
* when the source has no explicit intro but the surrounding text names the output signal, the canonical intro is synthesized from that context
* descriptive preambles (e.g., "This Output Indication shall present the moving-state status to the user.") are omitted per §8.6
* numeric value annotations like `(0)`, `(1)`, `(254)` are stripped -- use enum names only per §8.4
* when both raw values and engineering values (datatype constant names) appear in a table, always use the engineering value (datatype name) per §8.4 -- raw numeric values are only used when no datatype constant is defined
* "shall be created according to" passthrough tables (Input|Output|Supplement with "Carry over") are NOT table assignments -- they normalize to simple unconditional assignment (§4.1)
* the table must contain at least a header row + separator + 1 data row
* non-priority tables with overlapping row conditions must be disambiguated by adding exclusion guards (`AND`) -- do not introduce implicit priority
* priority tables (source explicitly states priority) get a `Priority` column with 1...n -- overlapping conditions are permitted because row order is semantically binding

**Boundary**

* this primitive does NOT cover data-dictionary / capability-matrix tables where the table defines what *exists* rather than a mapping from input to output
* this primitive does NOT cover tables embedded in stateful computation, timer lifecycle, or workflow contexts -- those belong to their respective layers
* this primitive does NOT cover tables where row matching depends on retained state, prior values, or temporal history

### 4.11 Priority-ordered conditional assignment

**Description**

When the source explicitly declares that conditions are ordered by priority (first-match-wins) but does NOT use a tabular format, the CNL output uses `(Priority N):` prefixed When/then statements.

This is the prose-form counterpart to §4.10's priority table. Both encode the same semantics (first matching condition wins, overlapping conditions are permitted). The difference is surface form only: source has a table  use §4.10; source has prose/list  use §4.11.

**Canonical template**

```text
(Priority 1): When <condition_1>, then <target> shall be set to <value_1>.
(Priority 2): When <condition_2>, then <target> shall be set to <value_2>.
(Priority 3): When <condition_3>, then <target> shall be set to <value_3>.
Otherwise, <target> shall be set to <fallback>.
```

When to use this form

Use `(Priority N):` When/then when ALL of the following are true:

* the source explicitly declares priority ordering -- using keywords such as: "priority", "prioritized according to", "first satisfied condition is sufficient", "first condition fulfilled", "highest priority", "(1)...(2)...(3)", or equivalent
* the source does NOT contain a markdown table
* conditions may overlap (i.e., multiple conditions could be true simultaneously, and the source relies on ordering to resolve which one applies)

When NOT to use this form

* source has no priority keywords and conditions do not overlap use normal When/Otherwise (no prefix)
* source has no priority keywords but conditions overlap this is a spec error in the source; flag
```

````
    for review
* source contains a markdown table with priority  use §4.10 priority table form
* source says "X has priority over Y" as a note confirming already-non-overlapping When/Otherwise order  the note is informational; use normal When/Otherwise without prefix (the ordering is already unambiguous)

**Evaluation semantics**

* statements are evaluated top-to-bottom -- first matching condition wins
* `(Priority 1)` is the highest priority
* overlapping conditions are permitted -- priority number resolves ambiguity
* `Otherwise` is the fallback when no priority statement matches
* if no `Otherwise` is present and no condition matches, the output is undefined (spec error in source) -- unless the source specifies a retain-default (§4.4)

**Relationship to §4.10 (table-driven assignment)**

Both `(Priority N):` and §4.10 priority tables encode first-match-wins semantics. The rule is: **follow the source format**.

* source is a table  §4.10 priority table (with Priority column)
* source is prose, bullet list, or numbered conditions  §4.11 `(Priority N):` When/then
* NEVER convert between the two forms -- preserve the source structure
* if a prose list, bullet list, or numbered list does not make the logical connector explicit, the source is ambiguous -- do not normalize it by inventing `AND` or `OR`

> **Disambiguation:** This "follow the source format" rule governs ONLY the choice between §4.10 and §4.11. Tables that match a different primitive's canonical shape (e.g., bus passthrough signal-routing tables with Input | Output | Supplement columns) are consumed by that primitive's own rules, not by §4.10.

**Examples**

Priority-ordered fallback (source: "first satisfied condition is sufficient"):

```text
(Priority 1): When Signal_A = Active, then Signal_A shall be set to Field_A.
(Priority 2): When Signal_C = Active, then Signal_A shall be set to Field_B.
Otherwise, Signal_A shall be set to Field_A.
```

Signal-source priority selection (source: "prioritized according to (1)...(2)...(3)"):

```text
(Priority 1): When Signal_A is within Valid Signal Range, then Signal_B = Signal_A.
(Priority 2): When Signal_C is within Valid Signal Range, then Signal_B = Signal_C.
(Priority 3): When Signal_D is within Valid Signal Range, then Signal_B = Signal_D.
Otherwise, Signal_A shall be set to NotAvailable.
```

Priority ladder with explicit ordering (source: "priority for setting X shall be in the following order"):

```text
(Priority 1): When Signal_A = Signal_A, then Signal_B shall be set to False.
(Priority 2): When Signal_C = Signal_A, then Signal_B shall be set to Signal_D.
(Priority 3): When Signal_C = Signal_A, then Signal_B shall be set to Signal_D.
(Priority 4): When Signal_C = Error AND Signal_C = Error, then Signal_B shall be set to Error.
Otherwise, Signal_B shall be set to False.
```

**Boundary**

* this primitive does NOT cover "X has priority over Y" notes that merely confirm already-unambiguous ordering -- those are informational and do not require `(Priority N):` prefix
* this primitive does NOT cover temporal alternation / oscillation (e.g., "shall alternate between X and Y with frequency F") -- that requires a clock and is out of scope for Behavioral CNL
* this primitive does NOT cover priority ordering that depends on retained state, counters, or timer expiry
````

```
---

## 5. Behavioral CNL - Partially supported primitives

These primitives have a validated in-scope subset, but they also contain boundary variants that must not be normalized as if the whole family were frozen.

### 5.1 Arithmetic assignment

**Validated in-scope subset**

Direct current-input expressions using the already frozen arithmetic surface belong to Behavioral CNL.

**In-scope example**

* ‘Signal_A = Signal_B / 60‘ is fully behavioral because it is one immediate expression over a current input only.

**Out-of-scope boundary**

Arithmetic rows that depend on reset-reference values or remembered prior state are out of scope even when the surface syntax looks simple.

**Boundary examples**

* accumulated metric since reset depends on ‘Signal_A(Ref)‘
* derived metric since reset depends on stored reference state
* elapsed metric since reset depends on stored reference state
* total metric since reset depends on stored reference state

**Scope rule**

Arithmetic assignment is mixed:

* direct current-input expression subset: in Behavioral CNL
* reset-reference or remembered-state subset: out of Behavioral CNL

### 5.2 Complex conditional mappings

> **Note:** The in-scope subset of complex conditional mappings is represented using the table-driven assignment surface (§4.10) when the source contains a tabular structure. Non-tabular sources without overlapping conditions use When/then statements.

**Validated in-scope subset**

Large conditional expansions remain in scope when the semantics are still explicit current-state branching with immediate fallback.

**Examples**

* parameter fallback with explicit current-state conditions
* range ladder with explicit fallback
* OR-expanded mapping from discrete input states

**Boundary cases**

Complex conditional rows leave the frozen core when they depend on:

* hidden arbitration or local precedence not visible in the surface
* ambiguous or overlapping guards that are not semantically resolved
* source-memory fallback or retained-state semantics beyond target-local keep-last-value

**Scope rule**

Complex conditional structure is acceptable only while it remains explicit current-state branching. Complexity alone is not disqualifying; hidden execution behavior is.

### 5.3 Priority table evaluation
```

```
> **Note:** The in-scope priority subset is fully frozen in §4.10 (tabular form) and §4.11 (non-tabular prose form). This section documents only the boundary variants that remain partially supported -- cases where stateful or timer behavior wraps around an otherwise-valid priority structure.

**Validated in-scope subset**

Explicit ordered current-state first-match ladders are in scope when row order is semantically binding and the fallback is explicit.

**In-scope examples**

* standard threshold ladder with ordered ‘When‘ rules and ‘Otherwise‘
* explicit priority selection with visible fallback

**Boundary cases**

Priority tables move out of the frozen behavioral core when they also depend on:

* counters, timers, or queue processes
* retained previous values inside the table mechanism
* stateful hold behavior or lifecycle interactions around the table

**Boundary examples**

* embedded counter/timer behavior
* queue/timer semantics around a priority table
* mixed table and stateful surrounding behavior

**Normalization notes**

* row order encodes priority
* ‘dont care‘ cells are omitted from visible conditions
* catch-all rows normalize to ‘Otherwise‘

### 5.4 Timing-qualified behavioral rules

**Validated in-scope subset**

Timing stays in scope when it only qualifies a current-state condition or a bounded immediate action.

**In-scope examples**

* sustained-condition timing with ‘for more than‘
* named delay ‘has not yet passed‘
* bounded duration assignment

**Out-of-scope boundary**

Timing leaves the behavioral core when it becomes a process model.

**Boundary examples**

* counter incremented every second
* countdown timer behavior
* timer reset and lifecycle actions as first-class behavior

**Scope rule**

‘Timer ≥ X‘ or ‘Delay has not yet passed‘ may be behavioral conditions. ‘start -> evolve -> check -> reset‘, periodic increment/decrement, and countdown logic are not.

---

## 6. Candidate behavioral extensions

These primitives remain close to the behavioral model: deterministic, atomic, and current-input driven. They are not yet frozen because the transform surface itself remains semantically essential and not yet part of the frozen grammar.
```

```
### 6.1 Linearization / lookup conversion (partially frozen)

**Frozen subset**

Named-table conversion assignments are frozen as §4.8 (conversion assignment). This covers rows
     where the conversion table exists as a named Requirements Engineering Tool entity (Local
     Constant or named calibration) and the requirement either declares the conversion or assigns a
     converted value with standard validity/error gating.

**Frozen rows**

* ‘Signal_A shall be converted using Table_A.‘
* ‘Signal_B shall be converted using Table_A.‘
* gated assignment with named conversion + status fallback
* cross-reference to named conversion
* current-input table-determined output with explicit invalid fallback

**Remaining candidate subset**

The following linearization shapes are still candidates, not frozen:

* **Inline calibration tables without a named reference**: source rows in this subset define ‘
     InputQuantity_A‘-to-‘OutputQuantity_A‘ conversion tables inline. These have no named Local
     Constant in the Requirements Engineering Tool. Only their error/invalid branches are
     translatable today. If a named reference is added in the Requirements Engineering Tool, they
     become fully translatable with §4.8.

* **Piecewise interval formulas**: source rows in this subset define multi-interval piecewise
     formulas with per-interval calibration parameters. These are current-input and deterministic,
     but the multi-interval structure is more complex than a single ‘converted using‘ reference.
     With ‘*‘ now in the arithmetic surface, these could be expressed as conditional arithmetic once
      the formula is spelled out explicitly.

* **Explicit mathematical formulas**: source rows with inline formulas, including a multi-point
     conversion formula, spell out the conversion logic directly. These are translatable as
     arithmetic with ‘*‘, ‘+‘, ‘-‘, ‘/‘, but the source uses mathematical notation that flattens to
     verbose arithmetic in CNL. Presentation is a pipeline concern.

* **Overflow guards referencing table limits**: source rows in this subset use guards such as ‘
     higher value than the limit of ConversionTable_A‘. This references a property of the conversion
      table, which has no frozen syntax. The conversion assignment itself is translatable; the
     overflow guard needs either an explicit threshold placeholder or new syntax for table-range
     references.

**Validated conclusion**

The named-table subset is frozen (§4.8). Inline tables, piecewise formulas, explicit mathematical
     conversions, and table-range overflow guards remain candidates.

### 6.2 Deferred computational seam: scaled conversion with overflow guard

**Status**

Structured and plausible, but still under-evidenced for promotion.

**Representative row**

* scaling plus explicit representable-range / overflow handling

**Boundary note**

This seam remains outside the frozen behavioral core. Overflow/clipping semantics materially change
     the mechanism and should not be collapsed into plain arithmetic assignment.

### 6.3 Retention-boundary candidates

The completed ‘latest_last_value‘ family analysis confirms that several recurring retention-adjacent
      forms have been resolved. The validated source-memory fallback subset is now frozen as §4.9.
     The remaining candidates below must not be treated as already-frozen extensions.
```

````
#### 6.3.1 Source-memory fallback -- FROZEN as §4.9

The narrow validity-triggered source-memory fallback subset is now frozen in §4.9. This includes:

* assignment from source's last valid value when validity-triggered
* source substitution when validity-triggered
* initialization guard when no valid value exists
* previous-value condition guard when explicit and immediate

Shapes that remain outside §4.9 (not frozen):

* `last known value` that is NOT triggered by explicit validity/error condition
* `last received value` without validity gating
* source-memory embedded inside workflow, timer, or storage contexts
* reset-reference values or stored prior values with PersistentStore_A/storage semantics

#### 6.3.2 Previous-value guard (non-validity-triggered)

Candidate shape:

```text
When Signal_A = A AND last value of Signal_A = B, then Signal_C shall be set to C.
```

This remains a candidate only when the remembered-value condition is not a validity/error gate (which would be §4.9d), controls one immediate assignment, and the remembered-value condition remains explicit. It is not frozen because it must not be collapsed into ordinary current-value comparison.

#### 6.3.3 Fallback default after no previous value exists -- partially frozen

The validity-triggered form (`When no valid value of Source exists, then Signal_B shall be set to Signal_A`) is now frozen as §4.9c.

The non-validity-triggered form remains a candidate only:

```text
When no previous value of Signal_A is known, then Signal_B shall be set to Signal_A.
```

This remains a candidate/default-retention boundary when the trigger is not source-validity but rather general first-execution or initialization semantics without explicit storage.

---

## 7. Out-of-scope semantic families

These families are explicitly excluded from Behavioral CNL because they require state, time evolution, persistence, workflow, or multi-step computation.

### 7.1 Storage / Lifecycle semantics

Storage / Lifecycle semantics are out of Behavioral CNL because they depend on retained-state identity, persistent storage, restore/read behavior, write/store behavior, or storage-governed fallback.

These semantics belong to the separate Storage / Lifecycle CNL layer.

The frozen Storage / Lifecycle CNL primitives are:

* lifecycle-triggered store / restore
* stored-value initialization with default fallback
* guarded persistent store

The following remain real Storage / Lifecycle semantic families but are not frozen primitives in v1:

* backup-value synchronization
* invalid-input retention / restore policy
````

```
Behavioral CNL MUST NOT normalize storage or persistence behavior into stateless assignment rules.

The key boundary remains explicit:

* ‘Target shall keep its last value‘ is Behavioral only when it refers to the target’s own local last assigned value
* ‘Target shall keep its last valid value‘ is also Behavioral only when it refers to the target’s own local last valid assigned value and remains immediate, target-local, and non-storage-governed
* the narrow validity-triggered source-memory fallback subset is frozen Behavioral (§4.9); source-memory that depends on lifecycle-exit state or storage persistence belongs to Storage / Lifecycle
* non-validity-triggered previous-value guards remain candidates (§6.3.2), not frozen Behavioral retention forms
* PersistentStore_A, stored value, backup value, retained memory, and stored-value fallback belong to Storage / Lifecycle semantics rather than to the Behavioral core

### 7.2 Interface / control policy and workflow / protocol semantics

Interface/control behavior does not form part of Behavioral CNL.

**validated non-behavioral extension candidates**

* external interface exposure
* scheduled interface publication

**Representative rows**

* external interface capability exposure
* scheduled interface publication

**Workflow / protocol examples**

* procedural Signal_A notification flow

**Why out of scope**

These rows depend on interface policy, publication ordering, workflow sequencing, request/confirm behavior, or broader protocol semantics. They belong to future semantic modules, not to the behavioral core.

### 7.3 Workflow / Stateful semantics

Workflow / Stateful semantics are explicitly out of scope for Behavioral CNL.

Behavioral CNL remains strictly stateless and immediate.

Workflow / Stateful CNL captures control-process semantics that require explicit state, timer lifecycle, timeout-driven branching, or incremental accumulation.

These semantics require a separate execution model and are not reducible to stateless condition-action rules.

The two layers may share visible syntax where applicable, but they differ in execution semantics.

Behavioral CNL MUST NOT normalize workflow/stateful behavior into stateless rules.

### 7.4 Stateful computation families

Stateful computational families are fully out of scope.

#### Signal filtering / smoothing

**Representative rows**

* rolling average over time window
* low-pass filtering with sample-rate-driven dynamics

**Why out of scope**
```

```
These rows depend on previous samples, previous outputs, ramping, or other temporal dynamics. The effect is not instantaneous.

#### Incremental accumulator / integrator

**Representative rows**

* running accumulated metric updated every sample
* accumulated metric with previous-value retention and persisted storage

**Why out of scope**

These rows depend on prior value of the same accumulator, repeated cycle evolution, integration over time, reset references, or persistence. No behavioral subset was observed in the validated pool.

### 7.5 Validated algorithmic computation summary

The previous class-level ‘algorithmic computation‘ chapter is replaced by the following validated findings:

* ‘arithmetic assignment‘: mixed. Direct current-input expressions are in Behavioral CNL; reset-reference and remembered-state variants are out.
* ‘derived ratio / aggregate metric‘: resolved. The direct current-input subset is now covered by frozen arithmetic with ‘*‘ and ‘/‘. History-dependent and reset-reference variants are out of scope (source-memory).
* ‘linearization / lookup conversion‘: mixed. Named-table conversion subset is frozen (§4.8). Inline-table, piecewise formula, explicit mathematical conversion, and table-range overflow guard variants remain candidates or partial.
* ‘deterministic date/time transform‘: deferred. Only ~5 rows require actual date/time transform semantics (UTC-to-local, weekday derivation, epoch decomposition). Insufficient evidence for promotion. Most date/time signal rows are simple assignments or conditional mappings already covered by frozen primitives.
* ‘signal filtering / smoothing‘: fully out of Behavioral CNL.
* ‘incremental accumulator / integrator‘: fully out of Behavioral CNL.

No broader claim about ‘algorithmic computation‘ is valid. The validated boundary is primitive-specific.

### 7.6 Short non-normative notes

**Enum / type definition**

Rows in this subset define metadata rather than executable behavior. They are not behavioral rules and should not drive grammar design.

**Incomplete / ambiguous artifacts**

Rows in this subset remain outside the normalization core because their structure is ambiguous or materially incomplete. They are evidence of boundary, not candidates for grammar promotion.

---

## 8. Normalization and visible surface conventions

This section preserves the strong normalization guidance from earlier versions, but aligns it to the primitive-based structure used here.

### 8.1 Condition primitives

The frozen behavioral surface recognizes the following condition primitives:

* comparisons: ‘=‘, ‘≠‘, ‘>‘, ‘<‘, ‘≥‘, ‘≤‘
* bare identifier predicates
* ‘AND‘, ‘OR‘, ‘NOT‘, with parentheses
* explicit transitions: ‘Signal transitions from A to B‘
* range predicates: ‘Signal is within Valid Signal Range‘, ‘Signal is within Invalid Signal Range‘
* narrow received predicate: ‘Signal_A is received‘
* signal-missing predicate: ‘Signal is missing‘
* ‘abs(...)‘ comparisons
```

```
* ‘has not yet passed‘
* source-memory condition: ‘last valid value of Signal_A = Value‘ (§4.9d)
* source-memory existence: ‘no valid value of Source exists‘ (§4.9c)

The key constraint is semantic, not syntactic: these condition surfaces are allowed only when the row still denotes current-state evaluation rather than process evolution.

‘Signal_A is received‘ is valid only for event-received immediate assignment. It is not a generic frozen condition surface for all ‘received‘ wording.

‘Signal is missing‘ is a signal-availability state predicate -- it asserts that the signal is currently absent/unavailable on the bus. It is NOT the negation of ‘is received‘ (which is an event trigger). See §8.8 for the disambiguation rule between ‘Signal is missing‘ and ‘Signal = <missing-value>‘.

### 8.2 Temporal qualifiers and timing boundary

Frozen behavioral timing is limited to:

* ‘for more than Duration‘
* ‘for the duration of Duration‘
* ‘has not yet passed‘
* bounded duration assignment: ‘Signal_A shall be set to Value for Duration.‘

The following are not part of the frozen behavioral timing surface:

* ‘increment every second‘
* ‘decrease by 1 every second‘
* countdown timers
* timer start/reset/stop as first-class process behavior
* cycle-relative process semantics unless and until separately frozen

### 8.3 Core normalization rules

| Source form | Canonical visible form |
| --- | --- |
| ‘If‘ / ‘if‘ | ‘When‘ |
| ‘Signal_A‘ | ‘Otherwise‘ |
| ‘shall set to‘ | ‘shall be set to‘ |
| ‘Otherwise‘ | prefer ‘Otherwise,‘ in new output |
| missing trailing period | include trailing period in canonical output |
| ‘>=‘, ‘\geq‘ | ‘≥‘ |
| ‘<=‘, ‘\leq‘ | ‘≤‘ |
| ‘!=‘, ‘=!‘, ‘\ne‘ | ‘≠‘ |
| ‘==‘ | ‘=‘ |
| ‘When Signal_A = (A OR B)‘ | ‘When Signal_A = A OR Signal_A = B‘ |
| ‘(A OR B) = V‘ | ‘A = V OR Signal_A = V‘ |
| ‘(A AND B) = V‘ | ‘A = V AND Signal_A = V‘ |
| ‘Signal is Valid signal‘ variants | ‘Signal is within Valid Signal Range‘ |
| ‘set to valid values of X, otherwise keep last value‘ | explicit valid-range assignment plus explicit ‘Otherwise, Signal_A shall keep its last value.‘ |
| ‘Not Available‘ / ‘Other Value‘ as symbolic states | ‘NotAvailable‘ / ‘Other Value‘ |
| ‘linearized according to X‘ / ‘linearized according to the table X‘ | ‘converted using X‘ |
| ‘set to the linearized Source‘ | ‘set to Source converted using ConversionTable‘ (resolve from linked definition) |
| ‘consider Source to have its last valid value‘ / ‘considered to have their respective last known value‘ | ‘Signal_A shall be treated as its last valid value‘ (only when validity-triggered) |
| ‘last known value of Source‘ (validity-triggered only) | ‘last valid value of Source‘ |
| ‘If no previous value is known, presume X‘ | ‘When no valid value of Source exists, then Signal_B shall be set to X.‘ |
| ‘shall be created according to‘ (table intro) | ‘shall be set according to the following table:‘ |
| ‘Any other value‘ / ‘All other values‘ / ‘All other condition‘ / ‘All other cases‘ / ‘For all other conditions‘ / ‘Default‘ (table catch-all row) | ‘Otherwise‘ |
| ‘Signal missing‘ (table row, when not a datatype value) | ‘Signal missing‘ |
| ‘Any other value or signal missing‘ / ‘Other value or signal missing‘ / ‘Any other value or missing‘ / ‘All other values or signal missing‘ (table combined catch-all row) | ‘Otherwise or signal missing‘ |
| ‘dont care‘ / blank cell (table don’t-care) | ‘-‘ |
| ‘If Signal_A is missing‘ / ‘When Signal_A is missing‘ / ‘X goes missing‘ (when "missing" is NOT a datatype value) | ‘When Signal_A is missing‘ |
```

```
| ‘If Signal_A is missing‘ / ‘When Signal_A = missing‘ (when "missing" / "Missing" is a datatype value of Signal_A) | ‘When Signal_A = <exact_datatype_constant_name>‘ |

### 8.4 Enum value resolution (datatype-first)

When normalizing enum values, use the following resolution order:

1. **Datatype constant name (preferred):** Look up the signal’s datatype table in the Requirements Engineering Tool. If the datatype defines named constants (e.g., ‘Signal_B‘, ‘State_A‘, ‘Mode_A‘), use the constant name exactly as defined in the datatype.
2. **Requirement wording (fallback):** If the signal has no datatype constants (e.g., a numeric interval like ‘0..100‘), or the value in the requirement does not correspond to any defined constant, normalize the requirement’s wording using the surface normalization rules below.

Surface normalization for fallback values (when no datatype constant applies):

| Source form | Canonical form |
| --- | --- |
| ‘Not Available‘, ‘not available‘, ‘"Not Available"‘ | ‘NotAvailable‘ |
| ‘Other Value‘, ‘other value‘, ‘"Other Value"‘ | ‘Other Value‘ |
| ‘missing‘, ‘"missing"‘, ‘Missing‘ | ‘missing‘ |
| ‘Error‘, ‘error‘, ‘"Error"‘ | ‘Error‘ |

Do not include numeric enum values alongside constant names. For example, ‘Active (1)‘ in the source becomes ‘Active‘ in CNL, not ‘Active (1)‘.

Do not use quotation marks around enum values in CNL unless the value is truly a literal text string posted to a UI display.

### 8.5 Identifier and value conventions

* do not add articles before identifiers
* preserve signal and parameter names exactly when already valid identifiers
* preserve multi-word parameter names in visible CNL
* preserve ‘X of Y‘ references for subsignals
* preserve bus qualifiers on references
* prefer datatype constant names over requirement-local wording (see §8.4)
* prefer semantic enum names over numeric encodings
* keep quoted strings only when the source truly denotes literal text output

### 8.6 Exclusion rules

Exclude the following from normalized CNL unless they add operative behavioral content:

* descriptive preambles
* frame-routing headers
* explanatory notes
* parenthetical enum explanations
* wrapper text that only introduces a table or algorithm without adding behavioral logic

### 8.7 Unit handling

Strip engineering-unit annotations from signal comparisons unless the unit is part of the formal identifier. Keep unit text only when it is part of a visible duration surface such as ‘ms‘, ‘s‘, ‘min‘, or ‘seconds‘.

### 8.8 Signal-missing disambiguation ("‘is missing‘" vs "‘= <value>‘")

When the source requirement uses the word "missing" in a condition about a signal, the CNL must distinguish between two fundamentally different meanings:

| Meaning | CNL form | When to use |
| --- | --- | --- |
| Signal is not received / unavailable on the bus | ‘Signal is missing‘ | "missing" is **NOT** a defined datatype constant of the signal |
| Signal has the value "Missing" (a named enum constant) | ‘Signal = <exact_constant>‘ | "missing" / "Missing" **is** a defined datatype constant of the signal |

**Decision procedure:**
```

````
1. Look up the signal's datatype in the Requirements Engineering Tool.
2. Check whether the datatype defines a constant whose name is "missing", "Missing", "MISSING", or any casing variant that the source uses.
3. **If yes** -- the source is referring to a value match. Write: `Signal = <exact_datatype_constant_name>` (preserving the exact case from the datatype definition, per § 8.4).
4. **If no** -- the source is referring to signal unavailability. Write: `Signal is missing`.

**Examples:**

Source: "If Signal_A is missing, then Signal_B shall be set to NotAvailable."
- Signal_A's datatype has no constant called "missing"  **`When Signal_A is missing, then Signal_B shall be set to NotAvailable.`**

Source: "If Field_A = Missing, then output shall be set to Error."
- Field_A's datatype defines `Missing` as a datatype constant  **`When Field_A = Missing, then output shall be set to Error.`**

Source: "When Signal_A is Error, Not Available, or missing..."
- Signal_A's datatype has `Error` and `NotAvailable` as constants, but no "missing"  **`When Signal_A = Error OR Signal_A = NotAvailable OR Signal_A is missing`**

**In table cells:**

The same rule applies. `Signal missing` as a table-cell keyword means "signal not received" (same as `is missing` in a condition). If the source table has a row with the text "Missing" and the signal's datatype defines `Missing` as a constant, it is a normal value row -- not the `Signal missing` keyword.

**Relationship to `Otherwise or signal missing`:**

The table catch-all `Otherwise or signal missing` (§4.10) implicitly includes the `is missing` semantics -- it covers both unmatched present values AND signal unavailability. The `Signal missing` standalone table row is for when the source separates the availability fallback from the value fallback with different outputs.

**"goes missing" normalization:**

Source wordings like "X goes missing" or "signal goes missing" normalize to `X is missing` -- they describe the same availability state. The transition semantics ("goes") are not preserved because the CNL evaluates current state, not transitions into missing.

---

## 9. EBNF reference (frozen constructs only)

This section intentionally includes only the frozen Behavioral CNL surface.

```ebnf
document                    = statement , { statement } ;

statement                   = unconditional_statement
                            | when_statement
                            | otherwise_statement
                            | table_statement
                            | guarded_table_statement
                            | priority_when_statement ;

unconditional_statement     = signal_list , "shall" , unconditional_action , [ "." ] ;

unconditional_action        = [ "be" ] , "set" , "to" , value
                            | [ "be" ] , "converted" , "using" , identifier ;

when_statement              = "When" , condition , "," , [ "then" ] , assignment_list , [ "." ] ;

priority_when_statement     = "(" , "Priority" , number , ")" , ":" ,
                              "When" , condition , "," , [ "then" ] , assignment_list , [ "." ] ;

otherwise_statement         = "Otherwise" , [ "," ] , assignment_list , [ "." ] ;
````
````

```
table_statement             = target_list , "shall" , "be" , "set" , "according" , "to" , "the" , "following" , "table" , ":" , table_body ,
                              [ otherwise_statement ] ;

guarded_table_statement     = "When" , condition , "," , [ "then" ] ,
                              target_list , "shall" , "be" , "set" , "according" , "to" , "the" , "following" , "table" ,
                              [ { logical_and , assignment } ] , ":" , table_body ,
                              [ otherwise_statement ] ;

target_list                 = target , { ( "and" | "," ) , target } ;

table_body                  = table_header , table_separator , table_row , { table_row } ;
table_header                = "|" , column_name , { "|" , column_name } , "|" ;
table_separator             = "|" , "---" , { "|" , "---" } , "|" ;
table_row                   = "|" , cell_value , { "|" , cell_value } , "|" ;
column_name                 = identifier | "Priority" ;
cell_value                  = value | range_cell | comparator_cell | compound_cell
                            | passthrough_cell | dont_care | catch_all | priority_number
                            | keep_last_cell | keep_last_valid_cell ;
keep_last_cell              = "keep" , "its" , "last" , "value" ;
keep_last_valid_cell        = "keep" , "its" , "last" , "valid" , "value" ;
range_cell                  = value , "-" , value ;
comparator_cell             = comparator , value ;
compound_cell               = comparator_cell , { logical_and , comparator_cell } ;
passthrough_cell            = "=" , identifier ;
priority_number             = number ;
dont_care                   = "-" ;
catch_all                   = "Otherwise" | "Signal missing" | "Otherwise or signal missing" ;

assignment_list             = assignment , { logical_and , assignment } ;

assignment                  = set_assignment
                            | keep_last_value_assignment
                            | keep_last_valid_value_assignment
                            | source_memory_assignment
                            | treated_as_last_valid_assignment
                            | bounded_duration_assignment ;

set_assignment              = target , ( ( "shall" , [ "be" ] , "set" , "to" ) | "=" ) , value ;

keep_last_value_assignment  = target , "shall" , "keep" , "its" , "last" , "value" ;

keep_last_valid_value_assignment
                            = target , "shall" , "keep" , "its" , "last" , "valid" , "value" ;

source_memory_assignment    = target , ( ( "shall" , [ "be" ] , "set" , "to" ) | "=" ) ,
                              "last" , "valid" , "value" , "of" , identifier ;

treated_as_last_valid_assignment
                            = target , "shall" , "be" , "treated" , "as" ,
                              "its" , "last" , "valid" , "value" ;

bounded_duration_assignment = target , "shall" , [ "be" ] , "set" , "to" , value , "for" , duration ;

signal_list                 = target , { "," , target } , [ "," ] ;

condition                   = or_condition , [ temporal_qualifier ] ;
or_condition                = and_condition , { logical_or , and_condition } ;
and_condition               = not_condition , { logical_and , not_condition } ;
not_condition               = [ logical_not ] , comparison
                            | [ logical_not ] , "(" , condition , ")" ;

comparison                  = identifier , comparator , value
                            | identifier
                            | transition
                            | range_predicate
                            | received_predicate
                            | missing_predicate
```

```
                            | abs_comparison
                            | has_not_passed
                            | last_valid_value_comparison
                            | no_valid_value_predicate ;

transition                  = identifier , "transitions" , "from" , value , "to" , value ;

range_predicate             = identifier , "is" , [ logical_not ] , "within" , range_kind , "Signal"
      , "Range" ;

received_predicate          = identifier , "is" , "received" ;

missing_predicate           = identifier , "is" , "missing" ;

abs_comparison              = "abs" , "(" , identifier , ")" , comparator , value ;

has_not_passed              = identifier , "has" , "not" , "yet" , "passed" ;

last_valid_value_comparison = "last" , "valid" , "value" , "of" , identifier , comparator , value ;

no_valid_value_predicate    = "no" , "valid" , "value" , "of" , identifier , "exists" ;

temporal_qualifier          = "for" , "more" , "than" , duration
                            | "for" , "the" , "duration" , "of" , duration ;

duration                    = value ;

reference                   = cname , [ bus_qualifier ]
                            | cname , "of" , reference ;

target                      = reference ;
identifier                  = reference ;
named_value                 = reference ;

value                       = expr | quoted_string ;
expr                        = term , { ( plus | minus | star | slash ) , term } ;
term                        = atom ;
atom                        = named_value | number | "(" , expr , ")" | converted_value ;

converted_value             = identifier , "converted" , "using" , identifier ;

bus_qualifier               = "(" , "on" , cname , ")" ;

comparator                  = "≥" | "≤" | ">" | "<" | "=" | "≠" ;
logical_and                 = "AND" | "and" ;
logical_or                  = "OR" | "or" ;
logical_not                 = "NOT" | "not" ;
range_kind                  = "Valid" | "Invalid" ;
plus                        = "+" ;
minus                       = "-" ;
star                        = "*" ;
slash                       = "/" ;

cname                       = letter_or_underscore , { letter | digit | underscore | "." } ;
number                      = [ "+" | "-" ] , digit , { digit } , [ "." , digit , { digit } ] ;
quoted_string               = '"' , { any_character_except_quote } , '"'
                            | "'" , { any_character_except_apostrophe } , "'" ;
```

EBNF disambiguation notes

Within ‘cell_value‘ context (table cells), ‘-‘ is interpreted as ‘dont_care‘ when it appears alone, or as a range separator in ‘range_cell‘ (e.g., ‘0-100‘). It is never arithmetic minus inside a table cell. Arithmetic minus (‘minus‘) only appears within ‘expr‘ in assignment values.

The ‘catch_all‘ production in ‘cell_value‘ context matches the keywords ‘Otherwise‘, ‘Signal missing‘, or ‘Otherwise or signal missing‘. These are reserved phrases within table cells and cannot appear as signal names or values. The literal ‘*‘ is no longer used as a catch-all in table cells -- multiplication (‘star‘) only appears within ‘expr‘.

```
A table with a single catch-all (‘Otherwise‘) row and no other data rows is semantically equivalent to an unconditional assignment (§4.1) and should be normalized as such.

**EBNF boundary note**

Section 9 does not include candidate extensions, stateful computation, storage semantics, timer lifecycle processes, or workflow/protocol modules. Those are documented elsewhere in this specification so they do not get mixed into the accepted grammar.

**Grammar-alignment note on names and references**

The normative CNL preserves explicit Requirements Engineering Tool names as written where possible.

The grammar implementation must support event/signal/reference names that contain spaces, commas, hyphens, quotes, and similar source-name punctuation, as long as parsing remains unambiguous.

‘cname‘ in the EBNF is a simplified reference token and may need to be replaced or extended in ‘grammar.lark‘ by a broader name/reference token.

The event-received primitive should parse real event names without forcing unnatural identifier collapsing.

The simplified ‘cname‘ production is illustrative and does not fully define the implementation tokenization of Signal_A names. The implementation grammar may use a broader name token for references, provided it preserves the normalized CNL structure and avoids ambiguity around reserved keywords such as When, then, shall, AND, OR, is received, and shall be set to.

---

## 10. Provisional and deferred patterns

These patterns are validated as real boundary material, but they are not part of the frozen Behavioral CNL surface.

### 10.1 Validated but not frozen

* cycle-relative conditions such as ‘in the previous run cycle‘ / ‘in the current run cycle‘
* timed keep-last-value variants beyond plain target-local retained fallback
* scaled conversion with overflow guard
* inline-table linearization without a named Signal_A reference
* piecewise interval formulas and explicit multi-point interpolation formulas
* overflow guards referencing conversion table range limits
* deterministic date/time transforms (deferred; ~5 rows need actual transform semantics, rest are simple assignments)

### 10.2 Future semantic modules, not behavioral grammar

* external interface exposure
* scheduled interface publication
* broader workflow / protocol semantics
* storage / lifecycle semantics

### 10.3 Do not drive grammar promotion

The following remain observed boundary material, not grammar targets:

* time-ramped interpolation
* checksum / packing / bitfield computation
* edge-trigger toggle
* latched rolling-memory state
* enum tables and metadata artifacts
* structurally ambiguous or incomplete sources

Grammar promotion still requires repeated evidence, one stable meaning, behavioral compatibility, safe normalization, and supporting tests.

---

## 11. Behavioral CNL Finalization
```

```
### 11.1 Included in Behavioral CNL

The accepted behavioral core includes:

* unconditional assignment
* simple conditional assignment
* explicit ‘Otherwise‘ fallback
* validity-gated assignment with explicit valid/invalid range predicates
* target keeps its own last value: ‘Target shall keep its last value‘
* target keeps its own last valid value: ‘Target shall keep its last valid value‘
* multi-assignment with ‘AND‘
* event-received immediate assignment when it satisfies the frozen scope constraints in Section 4.6
* explicit comparisons, boolean composition, and explicit transitions
* sustained-condition timing: ‘for more than‘, ‘for the duration of‘, ‘has not yet passed‘
* bounded duration assignment
* explicit priority ordering when the rule still depends only on current inputs and first-match
    behavior
* table-driven assignment (§4.10): non-priority tables with independent rows, priority tables with
    first-match behavior, guarded tables, and multi-output tables -- only when row matching depends
     solely on current input conditions
* priority-ordered conditional assignment (§4.11): ‘(Priority N):‘ When/then for non-tabular sources
     with explicit priority ordering -- only when conditions depend solely on current inputs
* direct current-input arithmetic expressions using ‘+‘, ‘-‘, ‘*‘, ‘/‘ over current inputs
* conversion assignment: ‘Source converted using Table_A‘ and ‘Signal_A shall be converted using
    Table_A‘ when the conversion reference exists as a named Signal_A entity
* source fallback to the last valid value (§4.9): validity-triggered ‘last valid value of Source‘
    assignment, source substitution, initialization guard when no valid value exists, and previous-
    value condition guard -- only when the fallback is triggered by an explicit validity or error
    condition

### 11.2 Excluded from Behavioral CNL

The validated exclusions are:

* source-memory semantics beyond the frozen §4.9 subset -- including non-validity-triggered ‘last
    known value‘, reset-reference state, stored prior values, ‘last received value‘ without
    validity gating, and source-memory embedded in workflow, timer, or storage contexts
* non-validity-triggered previous-value guards such as ‘last value of Signal_A = Y‘ or ‘previous
    value of Signal_A = Y‘ used as a condition without explicit validity/error trigger
* timer lifecycle processes, counters, countdowns, and periodic increment/decrement semantics
* workflow / protocol sequencing and interface lifecycle behavior
* request/response, timeout monitoring, first-receipt gating, until-received regimes, retry/abort/
    send-until behavior, and other received families that require workflow/protocol semantics
* storage persistence and initialization from stored state
* signal filtering / smoothing
* incremental accumulator / integrator
* hidden timezone / Signal_B policy and live clock progression
* checksum/packing and similar implementation-shaped computation families

### 11.3 Deferred beyond the frozen core

The following remain outside the frozen grammar but are still documented as boundary cases:

* inline-table linearization without a named Signal_A reference
* piecewise interval formulas and explicit multi-point interpolation formulas
* overflow guards referencing conversion table range limits
* deterministic date/time transforms (only ~5 rows; deferred for insufficient evidence)
* scaled conversion with overflow guard
* future non-behavioral semantic modules for interface/control and storage

### 11.4 Final alignment rule

The grammar should align to Section 9 only.

Sections 5, 6, 7, and 10 exist to document validated boundary decisions so that:

* in-scope behavior is not under-specified
* out-of-scope behavior is not accidentally normalized as if it were behavioral
* other candidate and deferred extensions remain visible without being promoted prematurely
```

```
Event-received immediate assignment is frozen Behavioral CNL when it satisfies the scope constraints
      in Section 4.6. Other received families remain Behavioral normalizations, Workflow / Stateful,
      Storage / Lifecycle, deferred, or not safely translatable depending on the main meaning of the
      requirement.

In other words, Behavioral CNL is complete when it has a small accepted core, a clear grammar, and
     explicit exclusions for cases that should not be forced into it.
```

E

Workflow / Stateful CNL Specification

This appendix provides the full normative Workflow / Stateful CNL specification used as part of the thesis artifact. The specification records the layer boundary, frozen primitives, and execution semantics that support the layered grammar basis discussed in Chapter 4 and keeps the artifact self-contained inside the thesis.

```
# Workflow / Stateful CNL Specification

This document is the normative design specification for Workflow / Stateful CNL in `vbc-cnl`.

This specification is organized around the final accepted Workflow / Stateful primitives and a clear boundary:

* only the five final Workflow / Stateful CNL primitives are normative
* scope is determined by meaning, not by source class labels or superficial wording
* visible syntax is reused from Behavioral-style CNL where possible, but the interpretation rules are separate
* storage, persistence, computation, and other non-workflow families are excluded unless explicitly covered by one of the final primitives

---

## 1. Purpose

Workflow / Stateful CNL captures control and state behavior that cannot be reduced to stateless immediate rules without losing operative meaning.

Its purpose is to model requirements whose meaning depends on one or more of the following:

* explicit named state and state transition
* process progression across stages or steps
* timer lifecycle control
* timeout-conditioned branch selection or failure handling
Workflow / Stateful CNL is separate from Behavioral CNL because it models stateful and process-driven behavior rather than stateless immediate evaluation. It is not an extension of Behavioral CNL. It is a separate semantic layer that reuses compatible visible syntax where possible.

### 1.1 Latest/last/previous value boundary

Workflow / Stateful CNL owns rows where previous-value wording is embedded in temporal or process semantics rather than standing alone as an immediate target-local fallback.

Representative boundary shapes include:

* timed hold of previous value or last valid value
* delay regime that preserves a previous value
* until-new-value or temporary retained regime
* transition-triggered keep-last behavior
* timer-governed fallback or expiry-controlled previous-value handling
```

```
37
38 These rows must not be normalized as frozen Behavioral retained fallback even when they contain a
       surface clause such as ‘Otherwise, Signal_A shall keep its last value‘ or ‘Otherwise, Signal_A
       shall keep its last valid value‘.
39
40 Immediate target-local last-valid-value retention belongs to Behavioral CNL only when it is one
       direct fallback assignment and timing or process behavior does not govern the retention. If the
        same wording appears inside a timed hold, delay, timeout, transition, timer lifecycle, or
       workflow progression regime, routing belongs to Workflow / Stateful CNL or to Mixed /
       decomposable analysis.
41
42 Similarly, validity-triggered source-memory fallback (e.g., ‘when Source is invalid, use last valid
       value of Source‘) belongs to Behavioral CNL §4.9 only when no timer, workflow progression, or
       process evolution governs the retention. If the same source-memory wording appears inside a
       timed hold, state-machine transition, or workflow context, routing belongs to Workflow /
       Stateful CNL.
43
44 ---
45
46 ## 2. Scope boundary
47
48 ### 2.1 In scope
49
50 Workflow / Stateful CNL is in scope only for requirements whose essential meaning depends on one or
       more of the following:
51
52 * explicit named states (identified by state-node requirement names) and transitions between them
53 * event-triggered transitions, when the event controls a state transition rather than a plain
       immediate assignment
54 * timer lifecycle control through explicit initialization, start, stop, or reset behavior
55 * timeout-driven branching, status change, state transition, or other explicit outcome assignment
56 * incremental accumulation where the new value of a target depends on adding a delta to its own
       current value
57
58
59 ### 2.2 Out of scope
60
61 The following are outside Workflow / Stateful CNL:
62
63 * storage / persistence semantics, including Signal_A behavior, retained values across lifecycle
       transitions, backup synchronization, and restore-from-storage behavior
64 * stateful computation beyond simple additive accumulation, including rolling averages, Signal_B
       filters, recurrence equations, iterative updates with complex formulas, and windowed
       computation
65 * simple event-triggered assignments or payload copies with no workflow or process semantics
66 * structural state-machine shells or lifecycle declarations that do not themselves provide
       executable transition semantics
67
68 ---
69
70 ## 3. Relationship to Behavioral CNL
71
72 Workflow / Stateful CNL reuses Behavioral-style visible syntax where possible, especially the
       ordinary ‘When ..., then ...‘ statement surface.
73
74 The distinction between Behavioral CNL and Workflow / Stateful CNL is semantic, not syntactic.
75
76 Behavioral CNL captures stateless, immediate condition-action rules. Workflow / Stateful CNL
       captures stateful, process-driven semantics whose meaning depends on explicit state, temporal
       process role, interaction correlation, or ordered progression.
77
78 Reuse of visible syntax does not imply reuse of Behavioral CNL scope. Workflow / Stateful CNL
       remains a separate execution model.
79
80 ### 3.1 Received-family boundary
81
82 Simple event-received immediate assignment now belongs to Behavioral CNL, not Workflow / Stateful
       CNL.
83
84 Example:
```

````
85
86  ```text
87  When Event_A is received, then Field_A of Signal_A shall be set to Value_A AND Field_B of Signal_A
        shall be set to Value_B.
88  ```
89
90  That surface remains Behavioral only when the received object is explicit and the result is one or
        more immediate deterministic assignments using existing Behavioral assignment forms, with no
        request/response correlation, timeout monitoring, first-receipt gating, receipt history,
        storage behavior, or protocol progression.
91
92  Workflow / Stateful CNL still owns received-related rows when the main meaning is process or state
        behavior, including:
93
94  * request received / response received correlation (deferred -- decomposes to §5.3 + Behavioral)
95  * matching response received after request
96  * confirmation received
97  * not received within a timeout
98  * missing-message monitoring
99  * first valid signal received or first-received gating
100 * until new values are received
101 * pending until a valid value has been received
102 * retry, abort, send-until, or protocol progression triggered by received or not-received conditions
103
104 Representative examples:
105
106 ```text
107 When Message_A has not been received within Timeout_A, then Status_A shall be set to Active.
        Otherwise, Status_A shall be set to Inactive.
108 ```
109
110 ```text
111 When Message_B is received that matches Message_A, then Result_A may proceed.
112 ```
113
114 ```text
115 Use previous/default behavior until new signal values are received.
116 ```
117
118 ```text
119 The operation fails if no matching values are received within Duration_A.
120 ```
121
122 Workflow / Stateful CNL must not absorb simple event-received assignment when no workflow or process
        semantics are required.
123
124 ---
125
126 ## 4. Semantic classification framework
127
128 Classification in Workflow / Stateful CNL is based on operative semantics, not on surface wording.
129
130 ### 4.1 Frozen v1 primitive
131
132 A frozen v1 primitive is one of the five normative semantic units defined in this specification. A
        requirement is safely classifiable inside Workflow / Stateful CNL only when its essential
        meaning is fully captured by one frozen primitive or by an explicit combination of frozen
        primitives.
133
134 ### 4.2 Compound row
135
136 A compound row is a source requirement whose semantics combine two or more frozen primitives in one
        requirement surface. Compound structure does not justify creating new primitives.
137
138 ### 4.3 Boundary / proxy row
139
140 A boundary or proxy row is a source requirement that is semantically adjacent to a frozen primitive
        but is not a clean canonical representative because of non-normalized surface, implicit source
        wording, missing explicit targets, or additional coupled semantics.
141
````

```
142 ### 4.4 Not safely translatable
143
144 A source requirement is not safely translatable when normalization would require invented signals,
        invented states, invented failure or abort targets, invented timers, or other hidden semantics
        not explicitly present in the source.
145
146 ### 4.5 Storage / lifecycle semantics
147
148 Storage / lifecycle semantics are requirements whose main meaning is retention, persistence,
        restoration, synchronization, or lifecycle-carried remembered value instead of control-process
        progression.
149
150 ### 4.6 Stateful computation
151
152 Stateful computation is behavior whose meaning depends on prior numeric state, repeated update
        evolution, rolling history, integration, filtering, or iterative computation rather than
        workflow progression. Simple additive accumulation (§5.5) is the one exception -- it is frozen
        because its execution model is minimal and well-bounded. All other stateful computation remains
         outside v1.
153
154 ### 4.7 Governing rule
155
156 Classification is determined by the main meaning of the requirement. Surface syntax alone must not
        be used to force a requirement into Workflow / Stateful CNL.
157
158 ---
159
160 ## 5. Frozen Workflow / Stateful CNL primitives
161
162 Only the following five primitives are normative in Workflow / Stateful CNL.
163
164 ### 5.1 Guarded named-state transition
165
166 **Semantic intent**
167
168 Guarded named-state transition captures behavior where a transition requirement defines the
        conditions under which the state machine moves from one named state (node) to another.
169
170 **Required elements**
171
172 * one explicit source state name
173 * one explicit target state name
174 * zero or more guard conditions constraining when the transition is allowed (init transitions may be
         unconditional)
175
176 **State name resolution**
177
178 The state name used in the CNL is derived from the state-machine context -- not from a signal's
        datatype enumeration value. Resolution order:
179
180 1. If the requirement text explicitly names the state (e.g. ‘If in state Inactive‘), use that name
        directly.
181 2. Otherwise, derive the state name from the requirement name (e.g. state nodes are named ‘State:
        Signal_A‘  ‘Signal_A‘; transitions are named ‘Transition T1: Inactive to Signal_A‘  source ‘
        Inactive‘, target ‘Signal_A‘).
182 3. If no explicit state name can be resolved from either source, the row is not safely translatable
        for this primitive.
183
184 **Canonical template**
185
186 Two canonical surfaces exist, selected by state-carrier type:
187
188 *Node-based state carrier* -- used when the state machine is structured as explicit state nodes and
        transition edges in the Requirements Engineering Tool (e.g. requirement nodes named ‘State:
        Inactive‘, ‘Transition T1: Inactive to Signal_C‘):
189
190 ‘‘‘text
191 When in state <source_state_name>
192 AND <guard_conditions>
193 [<temporal_qualifier>],
```

````
194 then state shall transition to <target_state_name>.
195 ```
196
197 *Signal-based state carrier* -- used when the state is tracked by a local variable or signal that is
        both guarded and assigned (e.g. `State_A`, `Signal_A`):
198
199 ```text
200 When <state_carrier> = <source_state>
201 AND <guard_conditions>
202 [<temporal_qualifier>],
203 then <state_carrier> shall be set to <target_state>.
204 ```
205
206 Selection rule:
207
208 * If the state machine context in the Requirements Engineering Tool uses explicit state nodes (
        requirement names `State: X`) and transition edges (requirement names `Transition T* X to Y`),
        use the node-based surface.
209 * If the state is tracked by a named signal or local variable that appears as both a guard condition
        and an assignment target in the same requirement, use the signal-based surface.
210 * The signal-based surface is syntactically identical to Behavioral CNL assignment. The Workflow /
        Stateful classification is semantic, not syntactic -- it applies when the same carrier is
        checked in the condition and assigned in the action, representing a state transition.
211
212 When a signal-based state transition includes additional output assignments alongside the state-
        carrier update, the combined statement follows Behavioral CNL §4.5 (multi-assignment with `AND
        `). No decomposition is required because all assignments fire on the same condition.
213
214 Unconditional init transition:
215
216 ```text
217 State shall initially transition to <target_state_name>.
218 ```
219
220 Exit transition:
221
222 ```text
223 When <exit_trigger>, then <state_machine_name> shall exit.
224 ```
225
226 **Allowed forms / trigger modes**
227
228 * immediate trigger mode, where the transition occurs when the guard condition is satisfied
229 * event-edge trigger mode, where the guard contains an explicit transition event such as `Signal
        transitions from A to B`
230 * sustained-duration trigger mode, where the full condition is qualified by `for more than <duration
        >` or `for the duration of <condition>`
231 * unconditional init, where no guard exists and the transition fires on state-machine initialization
232 * exit, where an external lifecycle event (e.g. `LifecycleExit_A`) terminates the state machine;
        side-effect assignments on exit (e.g. setting an output to `Status_A`) are included as
        additional `AND` clauses when the source explicitly prescribes them
233
234 Event-edge trigger is not a separate primitive. It is a trigger mode of guarded named-state
        transition.
235
236 **Timer-status predicates as guard conditions**
237
238 Timer status predicates such as `<timer> has expired` may appear as guard conditions in this
        primitive when they are one of multiple conditions constraining the transition. The distinction
         from §5.3 (timeout-driven branch): if timer/timeout expiry is the sole decisive trigger
        selecting the outcome  §5.3. If timer status is one guard among several for a state transition
         §5.1 guard condition. Example: `When in state Signal_A_A AND Field_A = Not active AND Timer_A
        has expired, then state shall transition to Inactive.` -- the timer is one of three guards, not
         the sole decisive trigger.
239
240 Lost-communication predicates such as `<signal> has not been received within <duration>` may also
        appear as guard conditions. These express timeout-based absence detection as part of a compound
         trigger.
241
242 **Minimal IR**
````

````
243
244 ```json
245 {
246   "source_state": "string | null",
247   "target_state": "string",
248   "guard_conditions": ["condition"],
249   "trigger_mode": "immediate | event_edge | sustained_duration | mixed | init | exit",
250   "event_guard": "condition | null",
251   "duration_guard": "condition | null",
252   "exit_trigger": "condition | null",
253   "exit_side_effects": ["assignment | null"]
254 }
255 ```
256
257 **Exclusion boundary**
258
259 * timer lifecycle actions such as start, stop, reset, or initialize
260 * timeout-driven branching or failure handling
261 * request / response correlation (deferred -- see §8.1)
262 * multi-step workflow sequencing (deferred -- see §8.1)
263 * state-node output logic (see §5.4)
264
265 ### 5.2 Timer lifecycle control
266
267 **Semantic intent**
268
269 Timer lifecycle control captures explicit lifecycle manipulation of a named timer or counter-like
        lifecycle object under a guard condition.
270
271 **Required elements**
272
273 * one explicit timer or counter-like lifecycle object identifier
274 * one explicit guard condition
275 * one explicit lifecycle action
276 * one explicit action value when the lifecycle action is `initialize` or `start`
277
278 **Canonical template**
279
280 ```text
281 When <guard_condition>, then <timer_or_counter_like_lifecycle_object> shall <lifecycle_action_phrase
        >.
282 ```
283
284 **Allowed forms / lifecycle actions**
285
286 Allowed lifecycle actions are:
287
288 * `initialize`
289 * `start`
290 * `stop`
291 * `reset`
292
293 Allowed lifecycle action phrases are:
294
295 * `shall be initialized to <value>`
296 * `shall start counting from <value>`
297 * `shall stop counting`
298 * `shall be reset`
299
300 This is the only primitive-specific surface phrase extension in v1.
301
302 **Minimal IR**
303
304 ```json
305 {
306   "lifecycle_object_identifier": "string",
307   "guard_condition": "condition",
308   "lifecycle_action": "initialize | start | stop | reset",
309   "action_value": "string | null"
310 }
````

````
```

**Exclusion boundary**

* timeout-driven branching or timeout-conditioned outcome selection
* explicit named-state transition semantics
* request, confirmation, retry, abort, or sequencing semantics
* continuous time evolution such as periodic increase semantics
* rows where lifecycle manipulation is only one extracted fragment of a larger compound rule

**Normalization rules**

1. Combined stop-and-reset phrasing -- source texts frequently express stop and reset as
   a single compound phrase (‘shall stop counting and be reset‘,
   ‘shall be reset and stop counting‘).  These normalize to two lifecycle actions
   joined by ‘AND‘ on the same guard using Behavioral §4.5 multi-assignment:

   ```text
   When <guard>, then <timer> shall stop counting AND <timer> shall be reset.
   ```

2. Generic ‘shall be set to 0‘ on a timer object -- when the context is initial-value
   setup before counting begins, this normalizes to the lifecycle phrase
   ‘shall be initialized to 0‘.  When it appears after counting has occurred (clearing),
   it normalizes to ‘shall be reset‘.

### 5.3 Timeout-driven branch / failure transition

**Semantic intent**

Timeout-driven branch / failure transition captures behavior where timeout expiry is itself the
     decisive trigger for a branch, transition, status assignment, or other explicit outcome
     assignment.

**Required elements**

* one explicit timeout source such as a timer, counter, or elapsed-duration parameter
* one explicit timeout condition tied to threshold reach, expiry, or elapsed window completion
* one resulting branch, transition, status assignment, or other explicit outcome assignment selected
      because the timeout condition became true

**Canonical template**

```text
When <timeout_condition> [AND <context_guards>], then <outcome>.
```

**Allowed forms / timeout condition types**

Timeout condition may appear as:

* ‘<timer_or_counter> ≥ <threshold>‘
* ‘<timer> has expired‘
* ‘<guard_condition> for more than <duration>‘
* ‘<expected_event_or_result> is not satisfied for more than <duration>‘
* ‘<signal_or_message> has not been received within <duration>‘
* ‘<signal_or_message> has not been received within <duration> of <reference_event>‘

Timeout condition may also appear as missing-message monitoring or non-receipt-within-timeout, for
     example:

* ‘<signal_or_message> has not been received within <duration>‘
* ‘<signal_or_message> has not been received within <duration> of <reference_event>‘

The optional ‘of <reference_event>‘ suffix anchors the start of the timeout window to an explicit
     triggering event (e.g. ‘within 10 seconds of Signal_A‘). When omitted, the timeout window is
     continuous (message must arrive within every rolling window of the specified duration).

This form is in scope when timeout expiry is the branch-selecting condition and the resulting
     behavior is an explicit failure, status, output, or branch outcome.
````
````

````
374
375 Missing-message monitoring is a timeout-driven branch form, not a generic received rule. The
        operative meaning is that non-receipt within the explicit timeout selects the outcome.
376
377 **Canonical examples**
378
379 ```text
380 When Message_A has not been received within Timeout_A, then Status_A shall be set to Active.
        Otherwise, Status_A shall be set to Inactive.
381 ```
382
383 ```text
384 When Message_A has not been received within Timeout_B, then Status_A shall be set to Active.
        Otherwise, Status_A shall be set to Inactive.
385 ```
386
387 ```text
388 When Bus_A has not been received within Timeout_C, then Status_A shall be set to Active. Otherwise,
        Status_A shall be set to Inactive.
389 ```
390
391 ```text
392 When matching Message_B has not been received within Duration_A of Message_A, then Field_A of
        Signal_B shall be set to Status_B. Otherwise, Field_A of Signal_B shall be set to Status_C.
393 ```
394
395 Timeout semantics must be distinguished from ordinary temporal qualifiers. In this primitive,
        timeout expiry is the branch-selecting trigger. A temporal qualifier that only constrains how
        long another condition holds does not by itself create timeout-driven branching.
396 The timeout condition MUST be the decisive trigger for the outcome. Temporal qualifiers alone do not
         constitute timeout-driven branching.
397
398 This timeout-condition surface does not cover:
399
400 * simple event-received immediate assignment
401 * request/response correlation (deferred -- see §8.1)
402 * latest/last/previously received value
403 * first-receipt or until-received regimes
404 * storage or persistence behavior
405
406 Allowed outcome categories are:
407
408 * state transition
409 * output change
410 * status change
411 * branch handoff
412
413 Source rows that express failure or abort meaning without an explicit target signal or state name
        are boundary / provisional and are not canonical normalized examples of this primitive.
414
415 **Minimal IR**
416
417 ```json
418 {
419   "timeout_source": "string",
420   "timeout_condition_type": "threshold | expiry | elapsed_duration | timeout_window",
421   "timeout_reference": "string",
422   "timeout_window_anchor": "string | null",
423   "context_guards": ["condition"],
424   "outcome_type": "state_transition | output_change | status_change | branch_handoff",
425   "outcome_target": "string",
426   "outcome_surface": "assignment"
427 }
428 ```
429
430 **Exclusion boundary**
431
432 * pure timer lifecycle control without a timeout-driven outcome
433 * pure guarded named-state transition with no timer, counter, expiry, or elapsed-time trigger
434 * passive temporal qualification that does not itself select the branch or outcome
````

````
* source rows where no explicit timeout-conditioned outcome can be isolated safely

Timer lifecycle control remains distinct from timeout branching even when both appear in the same source requirement.

### 5.4 State-node output logic

**Status: frozen.**

**Semantic intent**

State-node output logic captures behavior that executes while the state machine is in a specific named state (node). Unlike guarded named-state transitions (§5.1) which define edges between states, state-node output logic defines what signals are produced or what values are assigned while the machine remains within a given state node.

State nodes range from simple (one unconditional output assignment) to complex (priority-table-driven output with timer-gated branching). This primitive covers the full range.

**Required elements**

* one explicit state name identifying the active node (resolved using §5.1 State name resolution)
* one or more output assignments scoped to that state

**State name resolution**

Same as §5.1: state names are derived from the requirement context, not from signal datatype enumeration values.

**Normalization of "set and keep"**

Source patterns such as ‘In state X: Set and keep signal Z = V‘ are normalized to ‘shall be set to‘. The "keep" semantics are implicit in the execution model -- a state-node output holds its value until the state machine transitions out or a higher-priority condition overrides it. No separate "keep" verb is needed.

**Canonical template**

Simple state-node output (continuous -- re-evaluated each cycle while in state):

‘‘‘text
When in state <state_name>,
then <output_signal> shall be set to <value>.
‘‘‘

State-node output with additional guards (e.g. timer state, priority conditions):

‘‘‘text
When in state <state_name>
AND <guard_conditions>,
then <output_signal> shall be set to <value>.
‘‘‘

Entry-triggered state-node output (one-shot -- evaluated once on state entry, latched):

‘‘‘text
When entering state <state_name>,
then <output_signal> shall be set to <expression>.
‘‘‘

The distinction between ‘in state‘ and ‘entering state‘:

* ‘When in state Signal_A‘ = continuous: the output logic is evaluated every execution cycle while the machine remains in state Signal_A.
* ‘When entering state X‘ = one-shot: the output logic is evaluated exactly once upon transition into state X, and the result is latched (held) until the machine transitions out.

Source patterns such as ‘When entering the state X, the values of A and Signal_A shall be read once‘ are normalized to the entry-triggered form. The "read once" / "shall be read once" wording maps to the one-shot execution semantics of the ‘entering‘ qualifier -- no separate "read once"
````

```
     verb is needed in the CNL surface.

When the state-node requirement contains a priority table or complex conditional logic, it reuses the same syntax forms as other CNL layers (Behavioral §4.6 priority evaluation, conditional mapping, etc.) -- prefixed with the ‘When in state <state_name>‘ or ‘When entering state <state_name>‘ guard. The priority-table syntax is not redefined here.

**Allowed forms**

* simple unconditional output assignment scoped to a state
* conditional output assignment with additional guards within a state
* priority-table-driven output selection within a state (reuses existing priority syntax)
* timer-gated behavior within a state
* edge-triggered side-effects (e.g. timer start) within a state
* entry-triggered one-shot assignment (latched until state exit)

**Minimal IR**

‘‘‘json
{
  "state_name": "string",
  "trigger": "in_state | entering_state",
  "guards": ["condition"],
  "output_assignments": [
    {
      "target": "string",
      "value": "string"
    }
  ]
}
‘‘‘

**Relationship to other primitives**

* §5.1 (Guarded named-state transition) defines edges -- when to leave a state. §5.4 defines what happens inside a state.
* §5.2 (Timer lifecycle control) -- timer actions within a state node may appear as compound behavior.
* Behavioral priority evaluation, conditional mapping, etc. -- the output logic within a state node reuses existing CNL syntax forms. The only addition is the ‘When in state <state_name>‘ context guard.

**Exclusion boundary**

* pure guarded named-state transitions (use §5.1)
* standalone timer lifecycle actions not scoped to a specific state node (use §5.2)
* state-machine shell declarations or containers with no executable behavior
* stateful computation (rolling averages, Signal_A filters, recurrence equations -- but see §5.5 for simple additive accumulation)

**Evidence**

* (State: State_A) -- simple: ‘Set and keep signal Signal_A = "State_A"‘  ‘When in state State_A, then Signal_A shall be set to State_A.‘
* (State: State_B) -- simple: ‘Set and keep signal Signal_A = "State_B"‘  ‘When in state State_B, then Signal_A shall be set to State_B.‘
* (State: State_C) -- complex: timer-gated priority table with edge-triggered timer start
* (State: State_D) -- complex: priority-table output for Field_A based on Signal_B signals

---

### 5.5 Incremental accumulator

**Semantic intent**

Incremental accumulator captures behavior where a named numeric target is updated by adding (or subtracting) a delta value to its own current value, triggered either by a discrete event or periodically at a stated interval.

The defining execution property is self-referential additive update: the new value depends on the
```

````
    old value of the same target (‘Target = Target + delta‘). This cannot be expressed by
    Behavioral CNL’s stateless assignment model.

**Required elements**

* one explicit accumulator target (counter, timer-as-counter, distance register, or named numeric
    variable)
* one explicit guard condition (when omitted, the accumulation is unconditional while the period
    applies)
* one explicit delta expression (the additive amount -- may be a constant, a signal, or a simple
    arithmetic expression)
* optionally, one explicit period qualifier specifying the accumulation rate

**Canonical template**

Event-triggered accumulation (fires once per trigger satisfaction):

‘‘‘text
When <condition>, then <accumulator> shall be incremented by <delta>.
‘‘‘

Periodic accumulation (fires repeatedly at the stated interval while the condition holds):

‘‘‘text
When <condition>, then <accumulator> shall be incremented by <delta> every <period>.
‘‘‘

Decrement forms (identical structure, reverse direction):

‘‘‘text
When <condition>, then <accumulator> shall be decremented by <delta>.
When <condition>, then <accumulator> shall be decremented by <delta> every <period>.
‘‘‘

**Allowed forms / accumulation constraints**

Allowed delta expressions include:

* integer constant (‘1‘, ‘100‘)
* calibration parameter name (‘Signal_A‘, ‘Timer_A‘)
* simple arithmetic expression with explicit operands (‘Signal_B - previous value of Signal_B‘ is
    NOT allowed -- see exclusion boundary)
* ‘the cycle time value in ms‘ (normalized form for Signal_C-period-based integration)

Allowed period qualifiers include:

* ‘every second‘
* ‘every <N> seconds‘
* ‘every minute‘
* ‘each cycle time‘
* ‘each <period_parameter>‘

When no period qualifier is present, the accumulation fires once per evaluation where the guard
    condition transitions to satisfied (event-triggered mode).

**Normalization rules**

1. Source variants such as ‘shall increase by 1 every second‘, ‘shall be increased by 1 each second
    ‘, ‘increases with 1 for every second‘ all normalize to ‘shall be incremented by 1 every second
    ‘.

2. Source variant ‘shall decrease by 1 each second‘ normalizes to ‘shall be decremented by 1 every
    second‘.

3. Source variant ‘Timer_A shall be set to Timer_B + 1 seconds‘ normalizes to ‘Timer_A shall be
    incremented by 1 every second‘ (self-referential assignment is accumulation).

4. Source variant ‘shall start counting from 0 AND shall increase by 1 every second‘ decomposes into
     two statements: one timer lifecycle initialization (§5.2: ‘shall be initialized to 0‘) plus
    one incremental accumulator (‘shall be incremented by 1 every second‘).
````

````
606
607 5. Source variant ‘shall count the number of Signal_A runcycles, starting from 0‘ normalizes to
        initialization (‘shall be initialized to 0‘) plus ‘shall be incremented by 1 each cycle time‘.
608
609 6. Ceiling/floor guards: source patterns such as ‘shall be incremented by 1 every second up until <
        limit>‘ normalize by placing the ceiling in the guard condition: ‘When <condition> AND <
        accumulator> < <limit>, then <accumulator> shall be incremented by 1 every second.‘
610
611 **Minimal IR**
612
613 ```json
614 {
615   "accumulator_target": "string",
616   "guard_condition": "condition",
617   "delta_expression": "string",
618   "direction": "increment | decrement",
619   "period": "string | null"
620 }
621 ```
622
623 **Exclusion boundary**
624
625 * rolling averages or windowed computation (‘average of last N samples‘)
626 * Signal_A filters or recurrence equations (‘y[n] = K*x[n] + (1-K)*y[n-1]‘)
627 * delta expressions requiring ‘previous value of <other_signal>‘ (retained-value territory, not
        simple accumulation)
628 * Signal_B persistence of the accumulated value (Storage/Lifecycle concern -- accumulation logic and
         persistence logic are separate primitives)
629 * complex arithmetic beyond simple additive delta (multiplication, division, formula-based
        computation)
630 * timer lifecycle actions (start, stop, reset) -- use §5.2; the timer counting itself may be
        expressed as accumulation but only when the source explicitly states increment semantics rather
         than lifecycle semantics
631 * continuous time evolution or physics-based integration (outside CNL scope)
632
633 **Relationship to other primitives**
634
635 * §5.2 (Timer lifecycle control) governs lifecycle commands (start/stop/reset/initialize). §5.5
        governs the numeric progression itself. When a source row contains both initialization AND
        increment, decompose into §5.2 (initialization) + §5.5 (increment).
636 * §5.3 (Timeout-driven branch) may use the accumulated value as its timeout condition (‘When <
        accumulator> ≥ <threshold>, then ...‘). The timeout branch and the accumulation are separate
        primitive statements.
637 * Behavioral assignment (‘shall be set to <value>‘) is absolute. Accumulator (‘shall be incremented
        by <delta>‘) is relative. They are never interchangeable.
638
639 **Canonical examples**
640
641 ```text
642 When Signal_D > State_D AND Signal_A = Active, then Timer_A shall be incremented by 1 every second.
643 ```
644
645 ```text
646 When Signal_A transitions from Signal_A > Threshold_A to Signal_A < Threshold_A, then State_A shall
        be set to Status_A.
647 When State_A = Status_A AND Signal_B = State_C, then Signal_B shall be incremented by Value_A.
648 ```
649
650 ```text
651 When Signal_A = True AND Signal_B = False, then Timer_A shall be incremented by 1 every second.
        Otherwise, Timer_A shall be set to 0.
652 ```
653
654 ```text
655 When Field_A = Active, then Signal_A shall be incremented by 1 every second. Otherwise, Signal_A
        shall be set to 0.
656 ```
657
658 ```text
659 When Signal_A = Active AND Timer_A < Timeout_A, then Timer_A shall be incremented by the cycle time
````

````
    value in ms each cycle time.
```

```text
When Timer_A ≥ Timer_B AND Signal_A = Signal_B, then Signal_B shall be incremented by 1 AND
    Signal_B shall be set to Signal_C AND Signal_C shall be set to 0.
```

---

## 6. Compound row handling

Many source requirements combine more than one Workflow / Stateful primitive in a single row.

Compound rows must be decomposed into multiple primitive interpretations when their operative
    meaning contains multiple distinct semantic units, such as:

* a guarded named-state transition plus timer lifecycle control
* timer lifecycle control plus timeout-driven branching
* timer lifecycle control plus guarded named-state transition
* timer lifecycle initialization plus incremental accumulator
* incremental accumulator plus event-triggered counter reset (Behavioral assignment)

The primitive set remains atomic. Compound structure does not justify creating additional primitives
    , meta-primitives, or blended families.

If a source requirement cannot be decomposed without inventing hidden targets, hidden states, or
    hidden workflow machinery, it is boundary / provisional or not safely translatable.

### 6.1 Self-contained statement rule

Each decomposed statement shall be syntactically self-contained: all operative guards and conditions
     must be explicit in each statement. Later statements in a compound decomposition must not rely
     on implicit state established by earlier statements without repeating the relevant conditions.

Example -- correct (self-contained):

```text
When Signal_A ≥ Threshold_A AND Signal_A = Off, then Signal_B shall be set to On AND Timer_A shall
    start counting from 0.
When Signal_A ≥ Threshold_A AND Signal_A = On AND Timer_A < Timer_B, then Signal_C shall be set to
    Active.
```

Example -- incorrect (implicit state dependency):

```text
When Signal_A ≥ Threshold_A AND Signal_A = Off, then Signal_B shall be set to On AND Timer_A shall
    start counting from 0.
When Timer_A < Timer_B, then Signal_C shall be set to Active.
```

The second example omits the speed threshold guard and the Triggered=On guard, creating ambiguity
    about when the statement is valid.

---

## 7. Normalization rules

Only the following Workflow / Stateful-specific normalization rules are normative in v1:

* reuse Behavioral-style condition and ‘When ..., then ...‘ surface where possible
* normalize ‘If‘ / ‘if‘ to ‘When‘
* use ‘shall be set to‘ for assignment-style outcomes
* use ‘≠‘ instead of ‘!=‘
* use lowercase ‘not‘
* preserve explicit state names exactly; do not invent state names, signals, timers, or hidden
    process state
* state names are derived from the requirement context (requirement text, requirement name, or state
    -machine container name) -- not from signal datatype enumeration values
````
````

```
* do not normalize ‘in state Signal_A‘ into invented signals or implicit states; preserve the explicit state name as-is
* normalize "set and keep" to ‘shall be set to‘; the "keep" semantics are implicit in the state-node execution model (value holds until transition or override)
* preserve signal and state names exactly as exposed by the source
* resolve enum values using the signal’s datatype table first; fall back to requirement wording with standard surface normalization when no datatype constant applies (see Behavioral CNL §8.4)
* do not include numeric enum values alongside constant names
* use ‘for more than‘ or ‘for the duration of‘ as the only normalized temporal-qualifier surfaces
* do not rewrite passive temporal guards into timer lifecycle logic
* keep timer lifecycle phrases distinct from ordinary assignment and restrict them to the four allowed lifecycle action phrases
* keep simple event-received immediate assignment in Behavioral CNL when no workflow/process semantics are present

* normalize state-transition phrasing: source patterns such as ‘If in state Signal_B ... then a transition to Signal_A shall be made‘ must be rewritten using explicit state names. For node-based state machines, normalized form: ‘When in state <source_state_name> AND <guards>, then state shall transition to <target_state_name>.‘ For signal-based state carriers, normalized form: ‘When <state_carrier> = <source_state> AND <guards>, then <state_carrier> shall be set to <target_state>.‘ Selection between the two follows the canonical template selection rule in § 5.1. State names come from the requirement node names (strip ‘State: ‘ prefix) or from the signal’s state values as used in the source, not from container labels.
* normalize state-output phrasing: source patterns such as ‘In state X: Set and keep signal Z = V‘ must be rewritten as conditional assignments guarded by the state context. Normalized form: ‘When in state <state_name>, then <output_signal> shall be set to <value>.‘ The "set and keep" surface is normalized to ‘shall be set to‘ because hold semantics are implicit in the execution model (output holds until next transition or override).
* normalize entry-triggered state-output phrasing: source patterns such as ‘When entering the state X, the values of A and Signal_A shall be read once‘ must be rewritten using the ‘When entering state <name>‘ guard. The "read once" wording is implicit in the entry-trigger execution model and does not appear in the CNL surface.
* bulk group assignment: source patterns such as ‘all signals in X, except Y, shall be set to V‘ are expanded during normalization into individual assignments for each signal in the group. The signal list is sourced from the signal group definition in the Requirements Engineering Tool. No bulk-set surface exists in the CNL -- each signal produces its own statement.
* normalize exit phrasing: source patterns such as ‘When LifecycleExit_A occurs, then StateMachine_A shall also exit‘ must be rewritten as ‘When <exit_trigger>, then <state_machine_name> shall exit.‘ If the source explicitly prescribes a side-effect on exit (e.g. a Note stating a signal shall become ‘Status_A‘), the side-effect is included only when it introduces actual logic or behavior; purely informative notes without prescriptive language are omitted.
* normalize missing-message timeout surfaces: source variants such as ‘has not been received for <duration>‘, ‘is not received within <duration>‘, ‘not received for more than <duration>‘, or ‘timeout of <duration>‘ (for non-receipt monitoring) shall all normalize to ‘has not been received within <duration>‘
* preserve source table row order: when the source requirement presents conditions in an explicit ordered table or numbered list, the CNL translation shall preserve the source order unless semantic analysis proves the conditions are mutually exclusive and order-independent

---

## 8. Explicit exclusions

Workflow / Stateful CNL explicitly excludes the following semantic families:

* storage / persistence semantics, including Signal_A behavior, restore semantics, retained-value synchronization, and remembered lifecycle state
* stateful computation beyond simple additive accumulation, including rolling averages, Signal_B filters, recurrence equations, windowed computation, and complex iterative numeric evolution
* simple event-received immediate assignment or payload forwarding with no workflow semantics
* implicit state-machine shell declarations or lifecycle containers with no executable transition rule

Rows centered on these semantics are outside Workflow / Stateful CNL even if they contain words such as ‘state‘, ‘timer‘, ‘received‘, or ‘memory‘.

### 8.1 Deferred patterns (v1.1 candidates)

The following patterns have been identified but are not frozen in v1. They are documented here for traceability.
```

````
751
752 **Timer expiry shorthand** -- Source patterns such as ‘<timer> has expired‘ or ‘<timer> is expired‘
        cannot be normalized to an explicit comparison because the source data uses the same name for
        both the elapsed-time concept and the threshold calibration parameter. No distinct variable
        names exist in the source for elapsed time vs. configured threshold. This pattern is left as a
        known ambiguity; rows using it are classifiable as boundary / provisional under §5.3 (timeout-
        driven branch) but cannot be canonically normalized without inventing variable names. Evidence:
         3 rows (Signal_A state machines).
753
754 **Stateful numeric accumulation** -- Frozen as §5.5 (Incremental accumulator). The simple additive
        sub-patterns (event-triggered counter and cycle-based integrator) are now normative. The
        following remain excluded from the frozen primitive and are deferred for v1.1 consideration:
        rolling averages, Signal_A filters, complex recurrence equations, calibration-lookup
        integration, delta expressions requiring retained previous values of other signals, and
        persisted accumulated values. Evidence for the remaining patterns: 7 rolling-average rows, 1
        filter row.
755
756 **Multi-step workflow / sequencing** -- Ordered progression semantics where later behavior is valid
        only because earlier progression has already occurred. Originally defined as a frozen primitive
         (§5.5 in earlier drafts). Demoted to a deferred pattern because: (a) no clean corpus evidence
        exists -- all 11 classified rows are either full interface/protocol specifications (excluded by
         the primitive’s own boundary) or misclassified; (b) it has no unique surface form -- it is
        syntactically identical to behavioral ‘When..., then...‘; (c) it reduces to compound §5.1 --
        ordered progression is already captured by a state machine with 2+ states where guard
        dependencies enforce ordering. Any requirement with genuine ordered steps already has explicit
        state-carrier structure that decomposes into multiple guarded named-state transitions. Rows
        that cannot be decomposed without inventing states are not safely translatable per §4.4.
757
758 **Request / response workflow** -- Correlation semantics where a request is issued, a matching
        response or confirmation is received, and the accepted correlation authorizes one resulting
        action. Originally defined as a frozen primitive (§5.4 in earlier drafts). Demoted to a
        deferred pattern because: (a) no clean corpus evidence exists -- all 10 classified rows are
        either full interface/protocol specifications (excluded by the primitive’s own boundary),
        misclassified received-event rows (pure Behavioral passthrough), or compound patterns mixing
        timeout + state + correlation; (b) it decomposes into §5.3 (timeout-driven branch for the
        failure path) + Behavioral (success path as conditional assignment); (c) the "correlation" is
        merely a guard condition (‘When <response> matches <request>‘), not a new primitive -- it uses
        existing Behavioral condition syntax. Rows with genuine request/response semantics that cannot
        be decomposed are not safely translatable per §4.4.
759
760 ---
761
762 ## 9. Minimal semantic IR reference
763
764 The following IR structures are normative primitive references. They must not be collapsed into one
        generalized super-IR.
765
766 ### 9.1 Guarded named-state transition
767
768 ‘‘‘json
769 {
770   "source_state": "string | null",
771   "target_state": "string",
772   "guard_conditions": ["condition"],
773   "trigger_mode": "immediate | event_edge | sustained_duration | mixed | init | exit",
774   "event_guard": "condition | null",
775   "duration_guard": "condition | null",
776   "exit_trigger": "condition | null",
777   "exit_side_effects": ["assignment | null"]
778 }
779 ‘‘‘
780
781 ### 9.2 Timer lifecycle control
782
783 ‘‘‘json
784 {
785   "lifecycle_object_identifier": "string",
786   "guard_condition": "condition",
787   "lifecycle_action": "initialize | start | stop | reset",
788   "action_value": "string | null"
````

````
}
```

### 9.3 Timeout-driven branch / failure transition

```json
{
  "timeout_source": "string",
  "timeout_condition_type": "threshold | expiry | elapsed_duration | timeout_window",
  "timeout_reference": "string",
  "timeout_window_anchor": "string | null",
  "context_guards": ["condition"],
  "outcome_type": "state_transition | output_change | status_change | branch_handoff",
  "outcome_target": "string",
  "outcome_surface": "assignment"
}
```

### 9.4 Incremental accumulator

```json
{
  "accumulator_target": "string",
  "guard_condition": "condition",
  "delta_expression": "string",
  "direction": "increment | decrement",
  "period": "string | null"
}
```

---

## 10. EBNF reference

The Workflow / Stateful CNL grammar is intentionally conservative. Most Workflow / Stateful primitives reuse the ordinary Behavioral-style statement surface. The distinction is semantic, not grammatical.

Only timer lifecycle control and incremental accumulator add primitive-specific action phrases.

### 10.1 Grammar delegation rule

Workflow / Stateful CNL reuses Behavioral CNL grammar.

The following nonterminals are defined in Behavioral CNL and are not redefined here:

* `behavioral_document`
* `behavioral_statement`
* `behavioral_when_statement`
* `behavioral_condition`
* `behavioral_value`

This specification extends only the semantic layer and introduces only minimal surface additions.

```ebnf
workflow_document = behavioral_document ;

workflow_statement = behavioral_statement
                   | timer_lifecycle_statement
                   | state_transition_statement
                   | state_node_statement
                   | accumulator_statement ;

condition = behavioral_condition
          | has_expired
          | not_received_within_predicate ;

has_expired                  = identifier , "has" , "expired" ;

not_received_within_predicate = identifier , "has" , "not" , "been" , "received" ,
````

```
                                "within" , value , [ duration_unit ] , [ "of" , identifier ] ;

(* §5.1 Guarded named-state transition *)
state_transition_statement =
    "When in state" , state_name ,
    "AND" , condition ,
    [ temporal_qualifier ] , "," ,
    "then state shall transition to" , state_name , "."
  | "State shall initially transition to" , state_name , "."
  | "When" , condition , "," , "then" , state_machine_name , "shall exit" ,
    [ "AND" , assignment ] , "." ;

(* §5.4 State-node output logic *)
state_node_statement =
    "When in state" , state_name ,
    [ "AND" , condition ] , "," ,
    "then" , assignment_or_priority_block , "."
  | "When entering state" , state_name ,
    [ "AND" , condition ] , "," ,
    "then" , assignment_or_priority_block , "." ;

state_name = identifier ;
state_machine_name = identifier ;

timer_lifecycle_statement =
    "When" , condition , "," , "then" , lifecycle_target , lifecycle_action_phrase , "." ;

lifecycle_action_phrase =
      "shall be initialized to" , value
    | "shall start counting from" , value
    | "shall stop counting"
    | "shall be reset" ;

lifecycle_target = identifier ;

(* §5.5 Incremental accumulator *)
accumulator_statement =
    "When" , condition , "," , "then" , accumulator_target , accumulator_action_phrase , "." ;

accumulator_action_phrase =
      "shall be incremented by" , delta_expression
    | "shall be incremented by" , delta_expression , "every" , period
    | "shall be decremented by" , delta_expression
    | "shall be decremented by" , delta_expression , "every" , period ;

accumulator_target = identifier ;
delta_expression = value | arithmetic_expression ;
period = value , time_unit | "cycle time" ;
time_unit = "second" | "seconds" | "minute" | "minutes" | "ms" ;

value = behavioral_value ;
‘‘‘

### 10.2 EBNF boundary note

The following primitive remains syntactically identical to Behavioral-style ‘When ..., then ...‘ statements:

* timeout-driven branch / failure transition

Their distinction is semantic only. This EBNF reference introduces no broad new grammar for them.

The EBNF extensions in this section are:

* the ‘has_expired‘ and ‘not_received_within_predicate‘ condition extensions
* the four allowed timer lifecycle action phrases (§5.2)
* the four allowed accumulator action phrases (§5.5)
* the ‘When in state <state_name>‘ guard form and ‘then state shall transition to <target>‘ action (§5.1)
* the ‘When <trigger>, then <state_machine_name> shall exit‘ exit form (§5.1)
```

```
* the state-node output statement form with ‘When in state‘ and ‘When entering state‘ variants (§
    5.4)
```

F

Storage / Lifecycle CNL Specification

This appendix provides the full normative Storage / Lifecycle CNL specification used as part of the thesis artifact. The specification records the layer boundary, frozen primitives, and storage semantics that support the layered grammar basis discussed in Chapter 4 and keeps the artifact self-contained inside the thesis.

```
# Storage / Lifecycle CNL Specification

This document is the normative design specification for Storage / Lifecycle CNL in `vbc-cnl`.

This specification is organized around the final accepted Storage / Lifecycle primitives and a clear boundary:

* only the three final Storage / Lifecycle CNL primitives are normative
* scope is determined by storage and persistence meaning, not by source class labels or incidental wording
* visible syntax is reused from Behavioral CNL where possible
* differences from Behavioral CNL are mainly about meaning, not syntax, except for a few explicit storage action phrases
* boundary and deferred families are documented only to keep the v1 core narrow and explicit

---

## 1. Purpose

Storage / Lifecycle CNL captures stored-value and persistence behavior that cannot be reduced safely to stateless immediate behavioral rules without losing operative meaning.

Its purpose is to model requirements whose essential meaning depends on one or more of the following:

* interaction with explicit retained or persistent storage
* initialization of a live value from stored state
* guarded persistence write behavior
* lifecycle-triggered storage synchronization

Storage / Lifecycle CNL is separate from Behavioral CNL because Behavioral CNL is stateless and immediate, while this layer depends on stored-state identity, persistence operations, or storage-governed fallback.

Storage / Lifecycle CNL is separate from Workflow / Stateful CNL because this layer is not about process progression, timer control, timeout branching, or request/response behavior. Lifecycle wording matters here only when it governs storage synchronization.

---

## 2. Scope boundary

### 2.1 In scope

```

```
Storage / Lifecycle CNL is in scope only for requirements whose essential meaning depends on one or more of the following:

* explicit read from retained or persistent storage
* explicit write to retained or persistent storage
* explicit initialization from stored value
* explicit default fallback determined by stored-value validity, accessibility, or absence
* explicit lifecycle-triggered synchronization between live state and retained state

### 2.2 Out of scope

The following are outside Storage / Lifecycle CNL:

* ordinary target-local retained fallback such as ‘Target shall keep its last value‘, which remains Behavioral CNL
* workflow or process progression semantics such as sequencing, state-machine progression, request/response correlation, retry, or confirmation behavior
* timer lifecycle control and timeout branching
* stateful numeric computation such as accumulation, filtering, rolling history, or iterative self-update
* interface or protocol readout semantics for stored values
* generic volatile memory buffering with no retained/persistent storage semantics
* full runtime architecture, storage subsystem design, or persistence infrastructure

### 2.3 Separation rule

Classification is determined by the main meaning of the requirement.

If storage or persistence is incidental to another dominant execution model, the row does not belong to Storage / Lifecycle CNL as a whole.

---

## 3. Relationship to Behavioral CNL

Behavioral CNL defines the general visible grammar.

Storage / Lifecycle CNL reuses Behavioral visible forms wherever possible, especially ordinary ‘When ..., then ...‘ statements and ordinary assignment structure.

The distinction between Behavioral CNL and Storage / Lifecycle CNL is primarily semantic, not syntactic.

Behavioral CNL captures stateless immediate condition-action behavior. Storage / Lifecycle CNL captures behavior whose meaning depends on explicit stored-state interaction, persistence write/read semantics, or storage-governed initialization and fallback.

No new general syntax is introduced in this layer. Only the minimal storage-specific action phrases listed in Section 8 are added as explicit surface forms.

### 3.1 Received-family boundary

Storage / Lifecycle CNL does not own received wording merely because the word ‘received‘ appears.

Storage / Lifecycle CNL applies only when the main meaning is explicit retained or persistent storage, restore/load behavior, guarded persistent store, or stored-value initialization.

Received-related rows may belong to Storage / Lifecycle CNL only when the source explicitly provides storage semantics, such as:

* a received value shall be stored in an explicit persistent storage object
* a stored value shall be restored or read on lifecycle entry
* a guarded persistent write to an explicit storage target shall occur

Source-memory phrases such as the following must not be normalized into Storage / Lifecycle CNL unless explicit persistent storage semantics are present:

* ‘latest received value‘
* ‘last received value‘
* ‘previously received valid value‘
```

````
* ‘use previous memory until X is received‘

These are source-memory or retained-state boundary cases. They belong to Storage / Lifecycle CNL only if the storage model and storage object are explicit. Otherwise they are routed as follows:

* validity-triggered source-memory (e.g., ‘when Source is invalid, use last valid value of Source‘) belongs to Behavioral CNL §4.9
* non-validity-triggered ‘last known value‘, ‘last received value‘, or ‘last value before lifecycle exit/timeout‘ must be preserved as-is and routed by semantic context -- to Storage / Lifecycle if storage is explicit, otherwise deferred or not safely translatable

Generic ‘Memory_A‘ or volatile buffering of cyclic received data is not Storage / Lifecycle v1.

Example:

```text
Cyclic data received via Signal_A shall continuously be stored in local buffer Memory_A.
```

This is volatile buffering and remains outside the current Storage / Lifecycle v1 frozen kernel unless the source explicitly defines retained or persistent storage semantics.

### 3.2 Latest/last/previous value boundary

Storage / Lifecycle CNL owns explicit persistent-store interaction involving previous, last, or last-valid values only when retained or persistent storage semantics are explicit.

Representative cases include:

* last valid value stored in PersistentStore_A
* previous or last-valid value restored from persistent storage
* lifecycle-triggered save or restore of remembered value state

These rows are not Behavioral retained fallback. Persistent store or restore of a last-valid or previous value belongs to Storage / Lifecycle CNL or to storage-boundary analysis that may require decomposition.

Target-local ‘keep its last valid value‘ is not Storage / Lifecycle by itself. It belongs to Behavioral CNL when it is an immediate target-local fallback with no explicit retained or persistent storage semantics.

Validity-triggered source-memory fallback (e.g., ‘when Source is invalid, use last valid value of Source‘) is not Storage / Lifecycle by itself. It belongs to Behavioral CNL §4.9 when there is no explicit persistent storage or storage-governed semantics. Storage / Lifecycle owns source-memory only when the retention mechanism is explicitly persistent (e.g., ‘last valid value before lifecycle exit‘, ‘value stored in PersistentStore_A‘).

By contrast, ‘Memory_A‘ or volatile latest-value buffering remains outside the frozen Storage / Lifecycle v1 kernel unless persistent semantics are explicit.

---

## 4. Semantic classification framework

Classification in Storage / Lifecycle CNL is based on operative semantics, not on wrapper prose or source labels.

### 4.1 Frozen v1 primitive

A frozen v1 primitive is one of the three normative semantic units defined in this specification. A requirement is safely classifiable inside Storage / Lifecycle CNL only when its essential meaning is fully captured by one frozen primitive or by an explicit combination of frozen primitives.

### 4.2 Compound row

A compound row is a source requirement whose semantics combine Storage / Lifecycle behavior with other semantic material such as workflow, computation, or additional storage families in one surface requirement.
````

````
141
142 ### 4.3 Decomposed semantic unit
143
144 A decomposed semantic unit is one extracted semantic part of a compound row. Storage / Lifecycle
      primitives may appear as decomposed semantic units inside compound requirements, but those
      compound occurrences must not widen primitive definitions.
145
146 ### 4.4 Boundary case
147
148 A boundary case is semantically adjacent to a frozen primitive but is not a clean canonical
      representative because of assignment-looking surface, remembered-source wording, generic memory
       wording, or additional coupled semantics.
149
150 ### 4.5 Not safely translatable
151
152 A row is not safely translatable when normalization would require invented storage objects, invented
       lifecycle triggers, invented targets, invented default values, or other hidden semantics not
      explicitly present in the source.
153
154 ### 4.6 Retained storage vs target-local retention
155
156 Retained storage means an explicit stored or remembered value that exists outside the target's own
      immediate local assignment history.
157
158 Target-local retention means the target keeps its own last assigned value. That remains Behavioral
      CNL and is not Storage / Lifecycle CNL.
159
160 ### 4.7 Governing rule
161
162 Surface syntax alone must not be used to force a row into Storage / Lifecycle CNL. Stored-state
      interaction or persistence semantics must be essential to the row's meaning.
163
164 ---
165
166 ## 5. Frozen Storage / Lifecycle primitives
167
168 Only the following three primitives are normative in Storage / Lifecycle CNL.
169
170 ### 5.1 Lifecycle-triggered store / restore
171
172 **Semantic intent**
173
174 Lifecycle-triggered store / restore synchronizes a live value with explicit retained storage at a
      lifecycle boundary.
175
176 Its main meaning is retained-state interaction, not process progression.
177
178 **Required elements**
179
180 * one explicit lifecycle trigger such as entering a lifecycle state, exiting a lifecycle state,
      lifecycle entry, lifecycle exit, or another explicit lifecycle boundary
181 * one explicit live target or live value
182 * one explicit stored source and/or stored destination such as ‘PersistentStore_A‘, a stored value,
      or ‘BackupStore_A‘
183 * one explicit read and/or write action
184
185 **Canonical templates**
186
187 ‘‘‘text
188 When <LifecycleEntry>, then <Target> shall be read from <StoredSource>.
189 ‘‘‘
190
191 ‘‘‘text
192 When <LifecycleExit>, then <Target> shall be written to <StoredDestination>.
193 ‘‘‘
194
195 ‘‘‘text
196 When <LifecycleEntry>, then <Target> shall be read from <StoredSource>.
197 When <LifecycleExit>, then <Target> shall be written to <StoredDestination>.
198 ‘‘‘
````

````
199
200 **Minimal IR**
201
202 ```json
203 {
204   "lifecycle_trigger": "string",
205   "lifecycle_phase": "entry | exit | lifecycle_entry | lifecycle_exit | other",
206   "target": "string",
207   "stored_source": "string | null",
208   "stored_destination": "string | null",
209   "action": "read | write | read_write",
210   "guard_condition": "condition | null"
211 }
212 ```
213
214 **Execution notes**
215
216 * evaluation is based on an observable lifecycle condition
217 * the read or write occurs atomically after evaluation
218 * no implicit storage object, lifecycle state, retained mirror, or synchronization logic may be
      introduced
219 * the lifecycle trigger does not represent workflow progression, protocol progression, or ordered
      multi-step behavior
220
221 **Exclusion boundary**
222
223 * ordinary assignment with no storage semantics
224 * workflow or process transitions where the lifecycle condition is part of process progression
      rather than persistence synchronization
225 * timer lifecycle control
226 * timeout branching
227 * request/response or protocol correlation behavior
228 * target-local retained fallback from Behavioral CNL such as ‘Target shall keep its last value‘
229 * computational state updates such as accumulators, rolling totals, filters, or iterative numeric
      state evolution
230
231 ### 5.2 Stored-value initialization with default fallback
232
233 **Semantic intent**
234
235 Stored-value initialization with default fallback initializes a live target from an explicit stored
      value source and assigns a default value only when the stored content is inaccessible, invalid,
       or missing.
236
237 Its main meaning is storage-dependent initialization, not process progression.
238
239 **Required elements**
240
241 * one explicit live target
242 * one explicit stored-value source such as ‘PersistentStore_A‘ or another explicitly named stored
      value
243 * one explicit validity or accessibility condition on the stored value
244 * one explicit default fallback value
245
246 **Canonical templates**
247
248 ```text
249 <Target> shall be read from <StoredSource>.
250 If <StoredSource> cannot be accessed or contains an invalid value, then <Target> shall be set to <
      DefaultValue>.
251 ```
252
253 ```text
254 When <InitializationContext>, then <Target> shall be read from <StoredSource>.
255 If <StoredSource> cannot be accessed or contains an invalid value, then <Target> shall be set to <
      DefaultValue>.
256 ```
257
258 The phrase ‘shall be read from <StoredSource>‘ is the canonical normalization of source expressions
      such as ‘shall have the value stored in <StoredSource>‘.
````

````
259
260 The context-based ‘When ..., then ...‘ form is allowed only when the source states an explicit
        initialization context.
261
262 **Minimal IR**
263
264 ```json
265 {
266   "target": "string",
267   "stored_source": "string",
268   "validity_condition": "condition",
269   "default_value": "Value_A",
270   "initialization_context": "condition | null"
271 }
272 ```
273
274 **Execution notes**
275
276 * evaluation depends on stored-value validity or accessibility
277 * the default fallback is applied only when the stored content is invalid or unavailable
278 * no implicit storage object, remembered source, or hidden validity condition may be introduced
279 * no lifecycle-triggered synchronization is implied unless the source states an explicit
        initialization context
280 * no workflow or process progression is implied
281
282 **Exclusion boundary**
283
284 * default assignment with no storage dependency, which remains Behavioral CNL
285 * lifecycle-triggered store or restore, which belongs to the lifecycle-triggered store / restore
        primitive
286 * guarded persistent store, which belongs to a separate Storage / Lifecycle primitive
287 * workflow or process semantics whose center is progression rather than storage-dependent
        initialization
288 * target-local retained fallback such as ‘Target shall keep its last value‘
289 * stateful numeric computation such as accumulators, rolling totals, or iterative value evolution
290
291 ### 5.3 Guarded persistent store
292
293 **Semantic intent**
294
295 Guarded persistent store writes a live source value into explicit retained storage only when an
        explicit guard is satisfied and suppresses the write otherwise.
296
297 Its main meaning is persistence write behavior controlled by a guard, not assignment to a live
        target and not process progression.
298
299 **Required elements**
300
301 * one explicit live source value
302 * one explicit stored destination such as ‘PersistentStore_A‘ or another explicitly named retained
        storage object
303 * one explicit guard condition controlling whether the write occurs
304 * one explicit non-store case when the guard is not satisfied
305
306 **Canonical templates**
307
308 ```text
309 When <WriteGuard>, then <SourceValue> shall be stored in <StoredDestination>.
310 When <NonStoreCondition>, then <SourceValue> shall not be stored in <StoredDestination>.
311 ```
312
313 **Minimal IR**
314
315 ```json
316 {
317   "source_value": "string",
318   "stored_destination": "string",
319   "guard_condition": "condition",
320   "action": "write"
321 }
````

```
```

The non-store case is implicitly defined as the negation of the guard condition unless explicitly stated in the source.

Execution notes

* evaluation depends on whether the explicit store guard is satisfied
* the persistent write occurs only when the guard permits it
* the non-store case means that retained state is not updated
* no implicit storage object, remembered value, or hidden default branch may be introduced
* no lifecycle-triggered synchronization is implied
* no workflow or process progression is implied

Exclusion boundary

* ordinary guarded assignment with no storage destination, which remains Behavioral CNL
* stored-value initialization with default fallback, which belongs to a separate Storage / Lifecycle primitive
* lifecycle-triggered store or restore, which belongs to the lifecycle-triggered store / restore primitive
* workflow or interface/protocol behavior whose center is confirmation, request handling, or session control rather than persistence write behavior
* target-local retained fallback such as ‘Target shall keep its last value‘
* stateful numeric computation such as accumulators, rolling totals, or iterative value evolution

---

6. Normalization rules

Storage / Lifecycle CNL uses the following storage-specific normalization rules.

* stored objects must be explicit; do not invent ‘PersistentStore_A‘, ‘BackupStore_A‘, remembered values, or retained destinations
* ‘memory‘ alone is not sufficient unless the source clearly indicates retained or persistent semantics
* lifecycle wording must not be interpreted as workflow progression when its only role is to control storage synchronization
* guards for guarded persistent store must control persistence itself, not merely the value assigned elsewhere
* fallback for stored-value initialization must depend on stored-value validity, accessibility, or absence
* received wording alone is not sufficient; explicit stored source/destination or explicit retained/persistent semantics are required
* source-memory phrases such as ‘latest received value‘, ‘last received value‘, ‘previously received valid value‘, or ‘use previous memory until X is received‘ remain boundary cases unless the storage model is explicit
* explicit ‘PersistentStore_A‘ store or restore of last-valid or previous values belongs here or to storage-boundary analysis; it does not revert to Behavioral retained fallback
* storage must be the main meaning of the requirement. If storage is incidental to another dominant behavior, such as computation, workflow, interface/protocol behavior, or local buffering, the row must not be classified as Storage / Lifecycle CNL as a whole
* preserve explicit storage objects, lifecycle triggers, targets, and default values from the source
* resolve enum values using the signal’s datatype table first; fall back to requirement wording with standard surface normalization when no datatype constant applies (see Behavioral CNL §8.4)
* do not include numeric enum values alongside constant names
* use only minimal storage-specific action phrases:
  * ‘shall be read from‘
  * ‘shall be written to‘
  * ‘shall be stored in‘
  * ‘shall not be stored‘
* compound rows must be decomposed into semantic units before classification
* compound evidence may support boundary understanding, but it must not widen frozen primitive definitions
* Storage / Lifecycle primitives may appear as decomposed semantic units inside compound requirements. Such occurrences may support classification, but they must not define or widen frozen primitive boundaries.

```
```

````
## 7. Explicit exclusions

The following are explicitly excluded from Storage / Lifecycle CNL:

* target-local retained fallback, which remains Behavioral CNL
* workflow or process semantics
* timer lifecycle control and timeout branching
* request/response or protocol correlation behavior
* stateful numeric computation including accumulators, filters, rolling history, and iterative updates
* interface or protocol readout of stored values
* generic volatile memory buffering or local cache semantics, including cyclic received data buffered only in `Memory_A` or volatile local attributes
* boundary/deferred families being used to widen frozen v1 primitive definitions

### 7.1 Boundary / deferred families

The following are real semantic families but are not frozen primitives in Storage / Lifecycle CNL:

* backup-value synchronization
* invalid-input retention / restore policy

These families must not be used to expand the three frozen v1 primitives.

---

## 8. EBNF reference

Storage / Lifecycle CNL reuses Behavioral CNL grammar.

This specification does not redefine the full grammar. It adds only minimal explicit storage-action phrases.

### 8.1 Grammar delegation rule

The following nonterminals are defined in Behavioral CNL and are not redefined here:

* `behavioral_document`
* `behavioral_statement`
* `behavioral_when_statement`
* `behavioral_condition`
* `behavioral_assignment`
* `behavioral_reference`
* `behavioral_value`
* `behavioral_logical_and`

```ebnf
storage_document = behavioral_document ;

storage_statement = behavioral_statement | storage_when_statement ;

storage_when_statement = behavioral_when_statement | storage_action_when_statement ;

storage_action_when_statement =
    "When" , condition , "," , [ "then" ] , storage_action_list , [ "." ] ;

condition = behavioral_condition ;

storage_action_list = storage_action , { logical_and , storage_action } ;

storage_action =
      read_from_action
    | written_to_action
    | stored_in_action
    | not_stored_action
    | behavioral_assignment ;

read_from_action =
    target , "shall" , "be" , "read" , "from" , stored_reference ;
````
````

````
written_to_action =
    target , "shall" , "be" , "written" , "to" , stored_reference ;

stored_in_action =
    target , "shall" , "be" , "stored" , "in" , stored_reference ;

not_stored_action =
  target , "shall" , "not" , "be" , "stored" , "in" , stored_reference ;

stored_reference = behavioral_reference ;

target = behavioral_reference ;
value = behavioral_value ;
logical_and = behavioral_logical_and ;
```

### 8.2 EBNF boundary note

Storage / Lifecycle semantics are primarily semantic constraints over Behavioral syntax, not new grammar structures.

This EBNF reference is limited to the minimal explicit storage action phrases:

* `shall be read from <StoredSource>`
* `shall be written to <StoredDestination>`
* `shall be stored in <StoredDestination>`
* `shall not be stored`

All additional expressions are constrained action phrases within the existing Behavioral CNL grammar shape.

No broad new syntax, hidden lifecycle phases, hidden memory constructs, or workflow/computation structures are introduced here.
````
````

G

Evaluation Survey

This appendix documents the survey used in Cycle IV and answered by 11 practitioners.

G.1 Section 1 — Participant background

What is your current role?

- Function Owner
- Logical Designer
- Software Developer
- V3 Tester
- V4 Tester
- V6 Tester
- Other

How many years experience do you have?

- 0-1 years
- 2-5 years
- 6-9 years
- 10+ years

How do you interact with requirements in your daily work? (Select all that apply)

- Write requirements
- Review requirements
- Test against requirements
- Implement from requirements
- Approve requirements

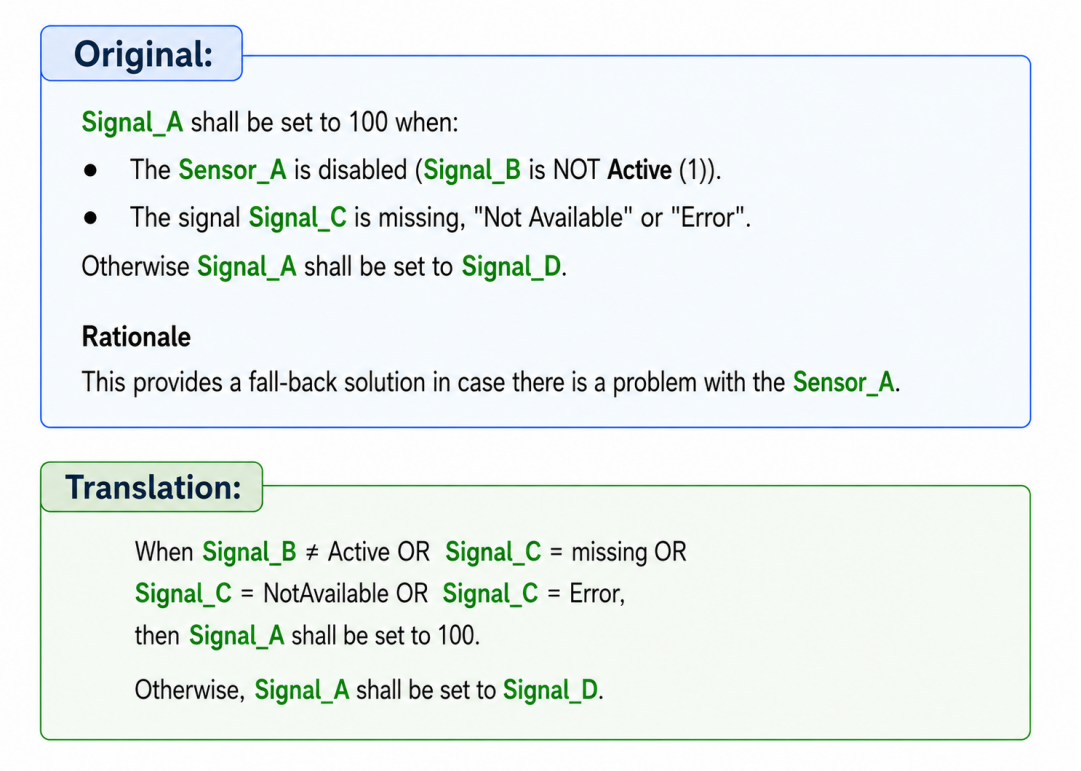
- Other

G.2 Section 2 — Requirement evaluation

Signal names, values, and identifiers in the requirements below have been replaced with anonymised placeholders such as `Signal_A` to protect proprietary information. The semantic structure and wording of each requirement are otherwise unchanged.

G.2.1 Requirement 1

This block contains the original requirement and its CNL translation.



Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

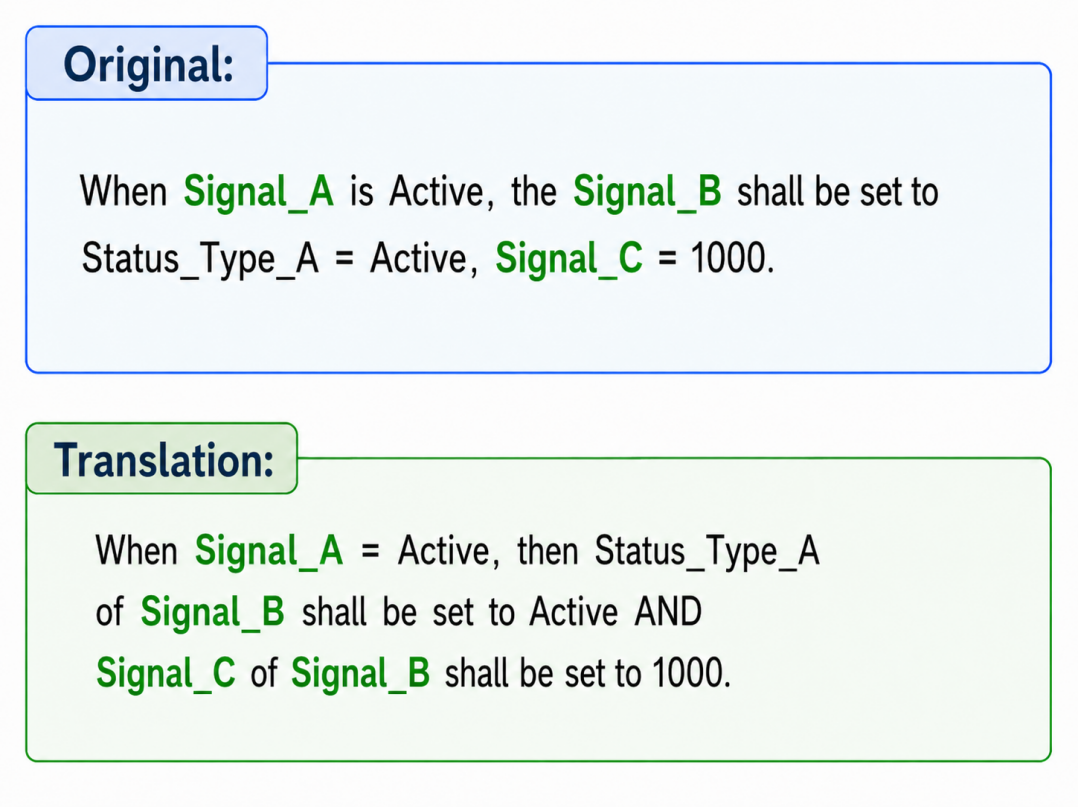
Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.2 Requirement 2

This block contains the original requirement and its CNL translation.



Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

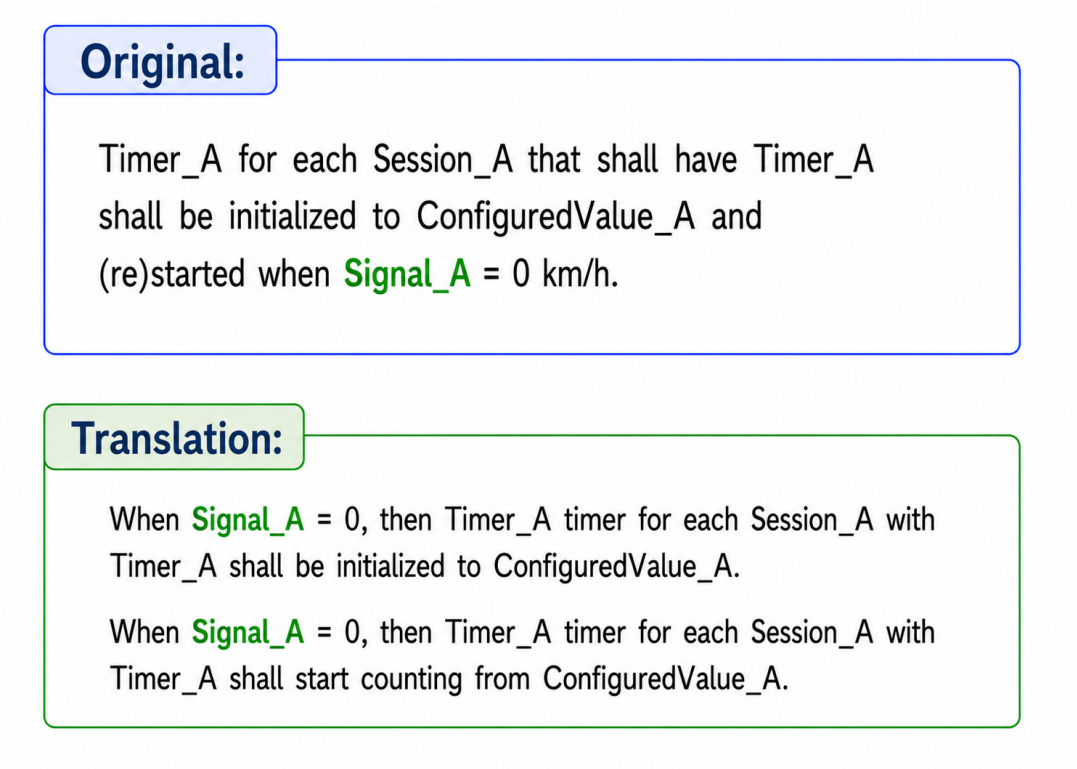
Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.3 Requirement 3

This block contains the original requirement and its CNL translation.



Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

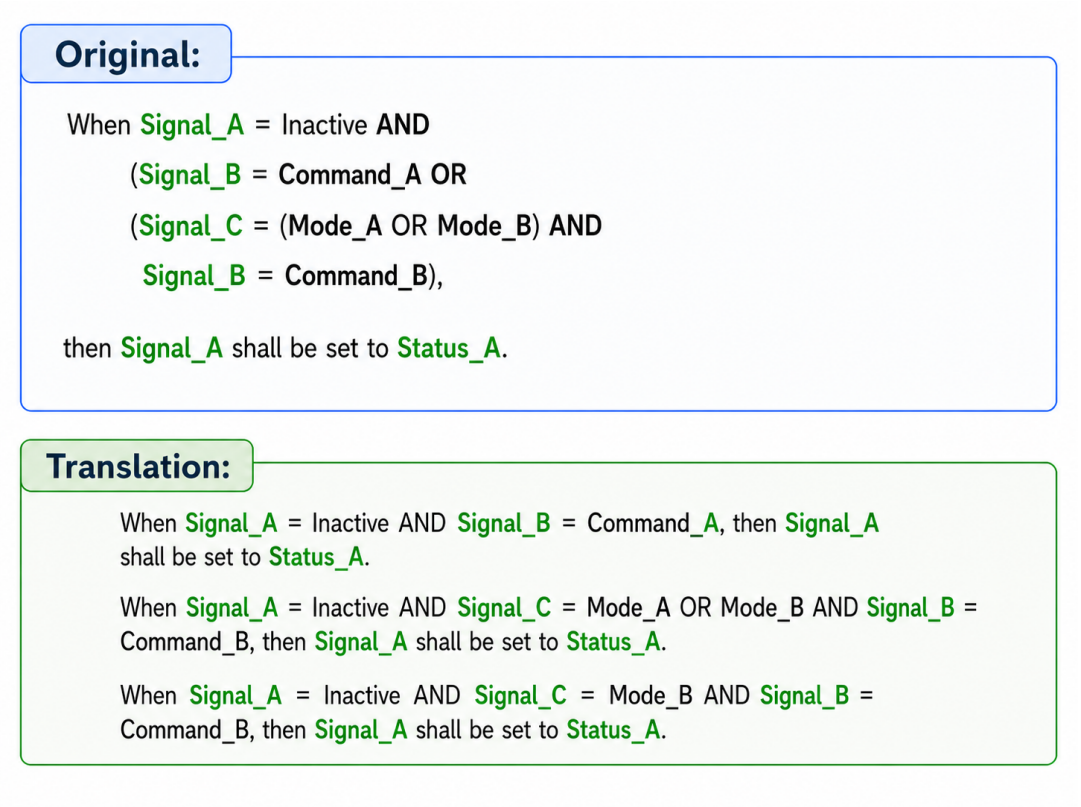
Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.4 Requirement 4

This block contains the original requirement and its CNL translation.



Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.5 Requirement 5

This block contains the original requirement and its CNL translation.

Original:

This Indication_A shall present warnings for **Resource_A** level.

The value of the signal Signal_A shall be used to set Signal_B according to the following table:

Signal_A		Signal_B	
Raw value	**Engineering value**	**Raw value**	**Engineering value**
0	Level 0	2 **	Moderate **
7	Not available	Keep previous value *	Keep previous value *
All other values		0	No Warning
Signal missing		Keep previous value *	

* If no previous value is known, presume 7: Not available
** Value shall only be set once per Cycle_A

Translation:

When Signal_A = Level0 AND Signal_B has not been set to Moderate in the current Cycle_A , then Signal_B shall be set to Moderate.

When Signal_A = Level0 AND Signal_B has been set to Moderate in the current Cycle_A , then Signal_B shall keep its last value.

When Signal_A = NotAvailable AND Signal_B has no last value, then Signal_B shall be set to NotAvailable.

When Signal_A = missing AND Signal_B has no last value, then Signal_B shall be set to NotAvailable.

When Signal_A = NotAvailable AND Signal_B has a last value, then Signal_B shall keep its last value.

When Signal_A = missing AND Signal_B has a last value, then Signal_B shall keep its last value.

Otherwise, Signal_B shall be set to NoWarning.

Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.6 Requirement 6

This block contains the original requirement and its CNL translation.

Original:

The value of the signal Signal_C shall be set according to the following table:

Signal_A	Signal_B	Signal_C
1 - On	Any value or signal missing	4 - Critical
Any other value or signal missing	1 - On	2 - Moderate
Any other combination of values or with signal missing		7 - Not available

Translation:

When Signal_A = On, then Signal_C shall be set to **Critical.**

When Signal_A ≠ On AND Signal_B = On, then Signal_C shall be set to **Moderate.**

Otherwise, Signal_C shall be set to **Not Available.**

Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

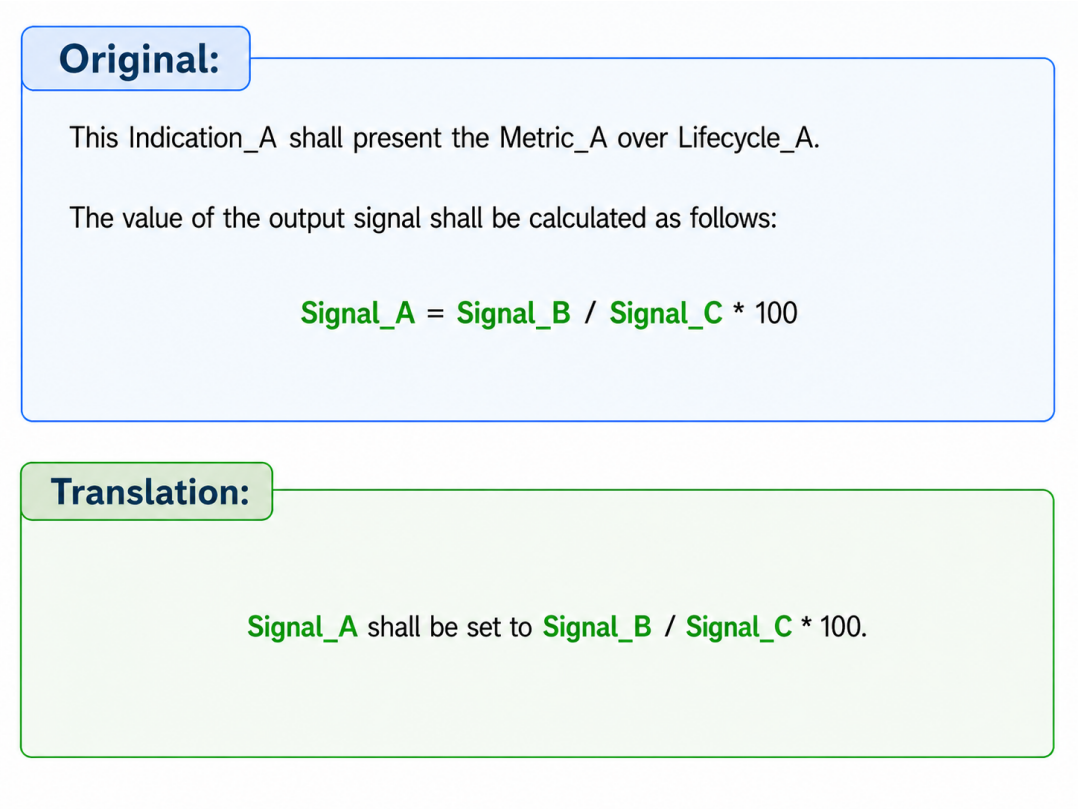
Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.7 Requirement 7

This block contains the original requirement and its CNL translation.



Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.8 Requirement 8

This block contains the original requirement and its CNL translation.

Original:

Signal_A shall be created according to:

When none of Signal_B, Signal_C, Signal_D are in Valid Signal Range,
Signal_A shall be set to NotAvailable.

When one or more of Signal_B, Signal_C, Signal_D are Valid Signal Range,
Signal_A shall be set to the one with highest priority.

The input signals shall be prioritized according to:
(1) Signal_B
(2) Signal_C
(3) Signal_D

Translation:

(Priority 1): When Signal_B is within Valid Signal Range, then Signal_A shall be set to Signal_B.

(Priority 2): When Signal_C is within Valid Signal Range, then Signal_A shall be set to Signal_C.

(Priority 3): When Signal_D is within Valid Signal Range, then Signal_A shall be set to Signal_D.

Otherwise, Signal_A shall be set to NotAvailable.

Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

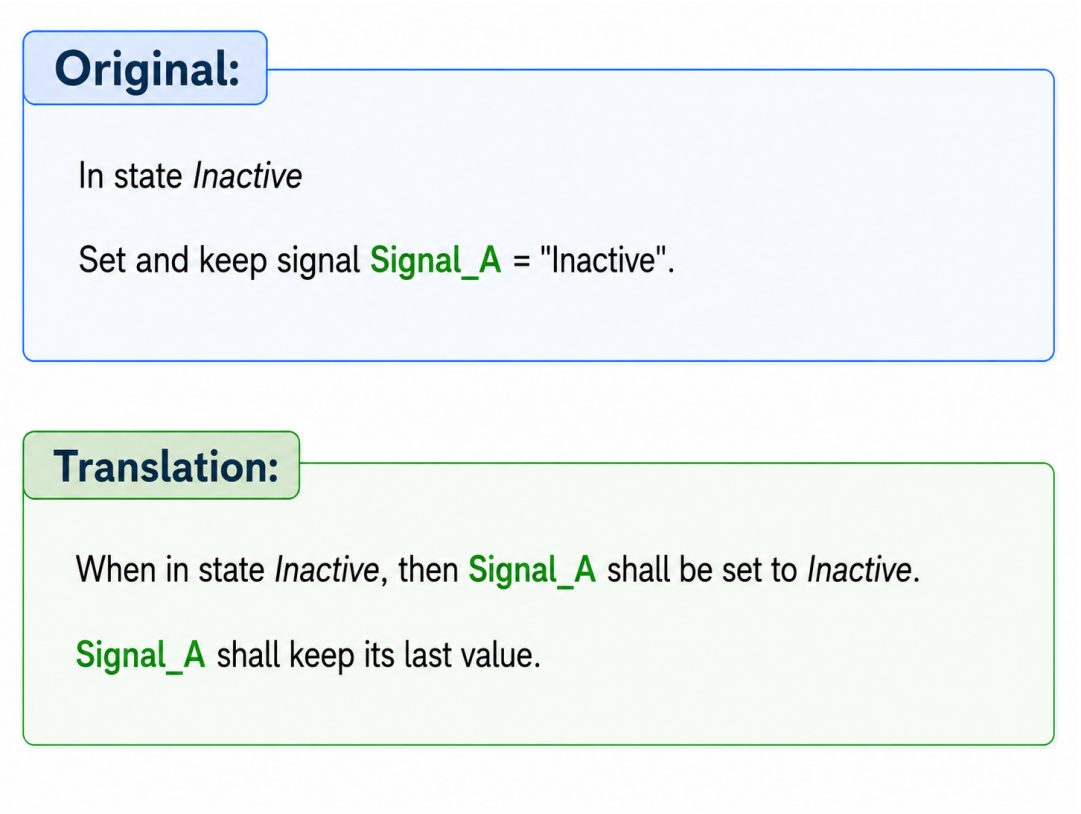
Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.9 Requirement 9

This block contains the original requirement and its CNL translation.



Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

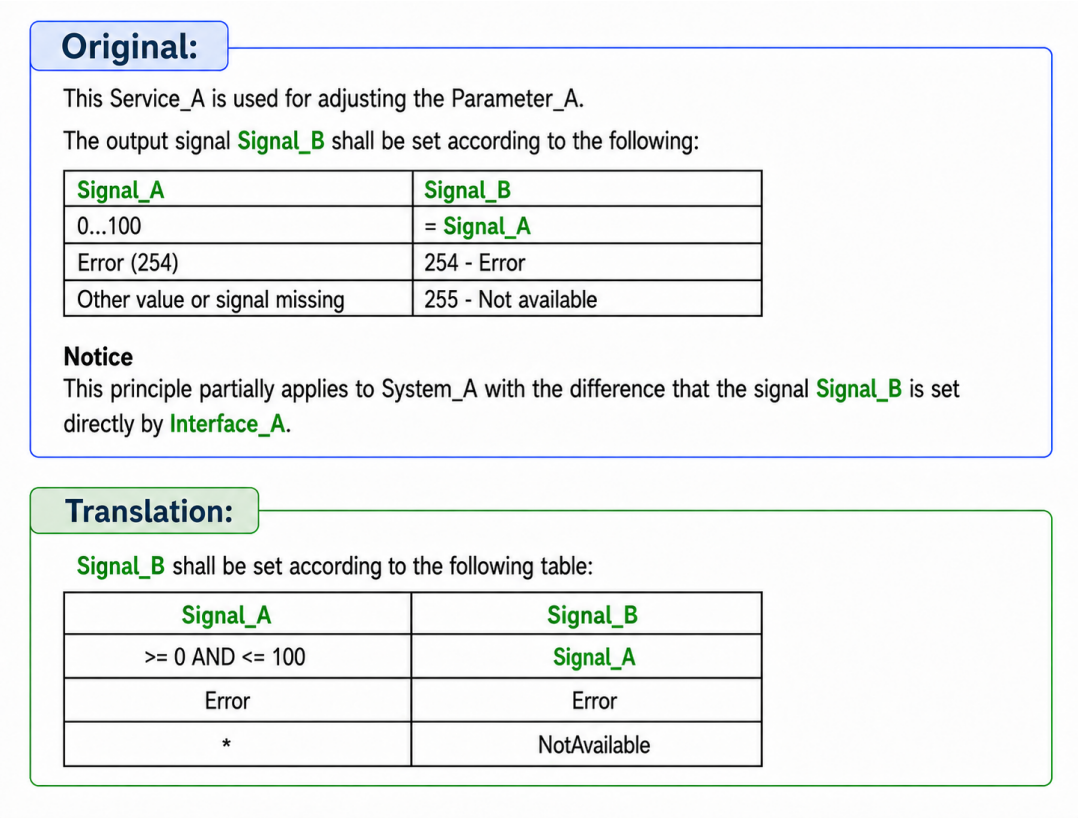
Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.2.10 Requirement 10

This block contains the original requirement and its CNL translation.



Original:

This Service_A is used for adjusting the Parameter_A.

The output signal Signal_B shall be set according to the following:

Signal_A	Signal_B
0...100	= Signal_A
Error (254)	254 - Error
Other value or signal missing	255 - Not available

Notice
This principle partially applies to System_A with the difference that the signal Signal_B is set directly by Interface_A.

Translation:

Signal_B shall be set according to the following table:

Signal_A	Signal_B
>= 0 AND <= 100	Signal_A
Error	Error
*	NotAvailable

Is the translation unambiguous

The rating scale runs from 0 to 10. The anchors are Very ambiguous and Very unambiguous.

Is the translation clearly written?

The rating scale runs from 0 to 10. The anchors are Not clear and Very clear.

Is the translation preferred over the original?

The rating scale runs from 0 to 10. The anchors are Definitely not preferred and Definitely preferred.

(Optional) Please provide a short explanation for your ratings.

G.3 Section 3 — Overall impression of the CNL translations

What is your overall impression of the Controlled Natural Language used in the translations?