UNIVERSITÄT ZU LÜBECK

Aus dem Institut für
Softwaretechnik und Programmiersprachen
der Universität zu Lübeck
Direktor: Prof. Dr. Martin Leucker

Formal Modeling and Analysis of Metric-Timed and Multi-Agent Normative Specifications of Actions

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck

Aus der Sektion Informatik/Technik

vorgelegt von
Karam Younes Kharraz
aus Rabat

Lübeck, 2026

1. Berichterstatter/Berichterstatterin:
(Name des/der Doktorvaters/-mutter erst eintragen nach Erhalt der Druckgenehmigung durchs MINT-Büro, d. h. nach der mündlichen Prüfung)

2. Berichterstatter/Berichterstatterin:
(Name erst eintragen nach Erhalt der Druckgenehmigung durchs MINT-Büro, d. h. nach der mündlichen Prüfung)

Tag der mündlichen Prüfung:
(erst eintragen nach Erhalt der Druckgenehmigung durchs MINT-Büro, d. h. nach der mündlichen Prüfung)

Zum Druck genehmigt. Lübeck, den _____

(erst eintragen nach Erhalt der Druckgenehmigung durchs MINT-Büro, d. h. nach der mündlichen Prüfung)

Declaration

Use of Generative Artificial Intelligence Tools Generative artificial intelligence tools were used during the preparation of this document as writing and editorial assistance. Specifically, OpenAI Chat-GPT, Google Gemini, and Grammarly (including multiple released versions of this tools) were used for (i) LaTeX programming assistance and (ii) language polishing, paraphrasing, and structural and content suggestions across the dissertation. No personal data, confidential information, or third-party copyright-protected materials were provided to these systems unless their use for this purpose was explicitly permitted. This disclosure follows the guidance from "Deutsche Forschungsgemeinschaft"[1] (DFG) and the University of Lübeck's[2] on the reporting of generative-model use in research outputs.

[1] https://www.dfg.de/resource/blob/289676/89c03e7a7a8a024093602995974832f9/230921-statement-executive-committee-ki-ai-data.pdf

[2] https://www.uni-luebeck.de/fileadmin/uzl_ki-uzl/pdf/20250525_KILeitlinieUzL_Web.pdf

Acknowledgments

TBD

Kurzfassung Normative Systeme bilden die wesentliche Infrastruktur, die das Verhalten von Individuen, Organisationen und autonomen Agenten regelt. Da diese Systeme in digitalen Umgebungen allgegenwärtig werden, wächst ihre inhärente Komplexität. Dies schafft einen kritischen Bedarf an einem rigorosen Engineering normativer Systeme. Ziel ist es sicherzustellen, dass Compliance nicht nur erreichbar, sondern auch einfach zu planen und zu verifizieren ist. Diese Dissertation nutzt formale Methoden, um dieser Herausforderung zu begegnen. Das Ziel besteht darin, formale Techniken so anzupassen, dass sie die von Regulierungsbehörden verwendeten Argumentationsprozesse automatisieren. Dadurch wird die Lücke zwischen mathematischer Logik und den praktischen Anforderungen der Compliance-Planung geschlossen.

Der erste Teil der Forschung konzentriert sich auf die statische Analyse von Spezifikationen. Er geht über die einfache Erfüllbarkeit hinaus, um normative Konflikte unter metrisch-zeitlichen Bedingungen zu untersuchen. Zu diesem Zweck etabliert die Dissertation die Micro Metric-Time Normative Logic (μMTNL). Dieser Formalismus ermöglicht eine präzise Unterscheidung zwischen *deontischen Konflikten* (widersprüchliche Regeln) und *ontischen Konflikten* (physikalische Unmöglichkeit). Unter Verwendung der Disjunctive Punctual Normal Form (DPNF) bietet das Framework eine rigorose Methode, um diese Inkonsistenzen zu erkennen. Zudem ermöglicht es Regulierungsbehörden, den Grad des Vorhandenseins von Konflikten über eine Metrik der „Konfliktdichte“ zu quantifizieren und diese systematisch zu eliminieren. Dies stellt sicher, dass die resultierenden Spezifikationen konfliktfrei sind und eine klare Grundlage für die Planung des Agentenverhaltens bieten.

Der zweite Teil adressiert die dynamische Überwachung normativer Interaktionen. Er konzentriert sich auf die Unterscheidung zwischen Versuchen und Erfolgen in kollaborativen Umgebungen. Jenseits der binären Erkennung von Vertragsverletzungen schlägt die Forschung die Two-Agents Collaborative Normative Logic (TACNL) vor, um komplexe Interaktionen zu modellieren. Ein zentraler Beitrag ist die Etablierung eines rigorosen, formalen Schuldzuweisungsverfahrens, das den Grad der Nichtkonformität für jeden Agenten über die gesamte Laufzeit eines Vertrags quantifiziert. Dieser Ansatz unterscheidet zwischen der Weigerung zu handeln und einem durch externe Störungen verursachten Scheitern. Folglich wird sichergestellt, dass die automatisierte Überwachung eindeutig, fair und umsetzbar ist.

Abstract Normative systems regulate the behavior of individuals, organizations, and increasingly, autonomous agents. The increasing structural complexity of normative specifications and the growing cost of ensuring their correct design and compliance motivate the need for rigorous Normative System Engineering based on formal methods. This dissertation explores how formal techniques can automate selected forms of reasoning commonly employed by regulators and normative experts.

The first part of the dissertation examines the static analysis of normative specifications under metric time. Instead of restricting the analysis to satisfiability, the focus is on identifying and characterizing normative conflicts. To achieve this, the Micro Metric-Time Normative Logic μMTNL is introduced. This logical framework enables a precise distinction between *deontic conflicts* (contradictory rules) and *ontic conflicts* (constraints on agent actions). By employing a Disjunctive Punctual Normal Form, the framework systematically detects such inconsistencies. Additionally, it introduces a quantitative measure of conflict density, enabling regulators to evaluate the prevalence of conflicts within a specification and eliminate them systematically. The outcome is a conflict-free specification that facilitates straightforward planning and execution by agents.

The second part of the dissertation addresses the dynamic analysis of normative interactions, particularly in collaborative contexts where compliance depends on both outcomes and agents' genuine attempts to fulfill obligations. To analyze these interactions, the Two-Agents Collaborative Normative Logic TACNL is developed, capturing collaborative norms and constructs such as reparations and Hohfeldian powers. Beyond binary violation detection, a blame procedure is introduced to quantify each agent's contribution to non-conformance throughout the duration of a contract. This procedure differentiates between deliberate inaction and failures resulting from interference or obstruction, ensuring that automated monitoring produces explanations and accountability assessments that are both precise and normatively significant.

Contents

1 Introduction

1.1 What are Normative Specifications?

A normative system is a regulatory framework that governs the behavior of autonomous agents by defining what is obligated, permitted, or forbidden. Unlike physical laws, which describe how a system *must* behave due to natural constraints, normative specifications prescribe how a system *ought* to behave, establishing a standard of correctness against which actions are judged. These specifications serve as the essential coordination mechanism in any organized environment, providing the explicit rules necessary to manage conflicts, ensure safety, and align individual goals with collective objectives.

A fundamental distinction within this domain lies between *ought-to-do* and *ought-to-be* specifications.

- **Ought-to-be (Seinsollen):** These norms describe ideal states of affairs without necessarily specifying the agent responsible for bringing them about. For example, a regulation stating that "the server room temperature must remain below 20°C" defines a required outcome. Violations are defined by the system state, but attributing responsibility requires additional context.
- **Ought-to-do (Tunsollen):** These norms prescribe specific actions to specific agents. For example, "the administrator must activate the cooling system if the temperature exceeds 19°C." Here, the focus is on agency and conduct.

While "ought-to-be" specifications are common in high-level policy, operational compliance and blame assignment rely heavily on translating these goals into precise "ought-to-do" rules that can be monitored against agent actions.

1.2 The Digitalization of Normative Systems

The digitalization of human activity is becoming pervasive, influencing nearly every aspect of daily life and societal function, ranging from payment systems and scheduling to the management of professional meetings and travel. Norms play a critical role in regulating these interactions, whether they occur between humans or between machines and

humans. As these interactions move into digital substrates, the need to represent, monitor, and enforce these norms computationally becomes acute.

1.2.1 The Cost of Compliance

The cost of compliance refers to the economic and operational burden that regulatory requirements impose on governments, businesses, and individuals. Beyond direct enforcement expenses, this burden also appears as administrative overhead, delays in decision making, reduced market access, and inefficiencies in organizational processes. Quantitative assessments suggest that these burdens can represent a substantial share of economic activity in highly regulated environments. For the European Union, the European Commission has reported an estimate that administrative burdens may amount to around 3.5% of GDP, based on an extrapolation approach and a specific definition of administrative burden.[1] For the United States, economy-wide estimates of the aggregate annual cost of federal regulation have been reported in the trillion-dollar range in the literature, with one widely cited estimate reporting roughly $2.0 trillion.[2] At the same time, official government reporting aggregates costs for a narrower subset of regulations with quantified impacts and therefore yields materially smaller totals, which highlights that published numbers differ substantially with scope and methodology.[3] At the firm level, regulatory costs per employee are often higher for small firms than for large firms, a pattern documented in analyses commissioned by the U.S. Small Business Administration.[4]

A large part of this burden does not arise from the substantive goals of regulation alone, but from its structural complexity. Normative systems are typically fragmented across jurisdictions, amended incrementally, and expressed in heterogeneous legal language. As a result, regulated entities must invest considerable effort in determining which obligations apply in a given situation, how they interact, and whether exceptions or compensatory clauses are triggered. This interpretive effort is repeated across organizations and over time, which leads to duplicated work and avoidable errors.

Operational consequences are particularly visible in high-volume administrative processes. Procurement, licensing, certification, and eligibility checks often require extensive manual review, even when decisions follow well-defined rules. Minor deviations or misin-

[1] https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX%3A52006DC0691

[2] https://www.congress.gov/crs-product/R44348 https://conservativereform.com/files/data-and-reports/cost-of-federal-regulations/federal-regulation-executive-summary.pdf

[3] https://bidenwhitehouse.archives.gov/wp-content/uploads/2025/01/FY23-Benefit-Cost-Report.pdf

[4] https://www.sba.gov/sites/default/files/The%20Impact%20of%20Regulatory%20Costs%20on%20Small%20Firms%20%28Full%29.pdf

terpretations can invalidate entire applications, wasting resources on both sides of the regulatory interface. From a systemic perspective, such processes scale poorly, since increasing regulatory detail can induce disproportionately higher verification and documentation costs.

These observations motivate the growing interest in computational approaches to compliance. Encoding regulations in machine-interpretable form enables automated reasoning about obligations, prohibitions, and permissions, as well as systematic detection of violations and identification of applicable remedies. By shifting compliance assessment from ad hoc interpretation to formal analysis, cost reductions become possible while simultaneously improving transparency, predictability, and legal certainty.

In this sense, formal methods do not merely offer technical optimization. They address a structural problem inherent in modern regulation, namely the gap between norm expression in natural language and the need for precise, repeatable, and auditable compliance decisions. Closing this gap is a prerequisite for scalable regulatory enforcement in increasingly complex socio-technical systems.

1.2.2 Efforts in the Digitalization of Compliance

The digitalization of normative systems is an active field of development. According to market analytics from Legal Complex, startups specializing in the digitalization of normative systems are present globally and have raised hundreds of millions of dollars in capital. Figure 1.1 illustrates results from the company's startup tracker focusing on the six largest global economies. A prominent example of success in this domain is Taxfix[5], a widely adopted software solution that has digitalized the tax declaration process in Germany.

To address the challenges of digitalization, the primary objective is to achieve a reasoning about norms as complete as possible while remaining strictly consistent. Two main branches of research address this goal:

1. Natural Language Processing (NLP) and its modern iteration, Large Language Models (LLMs).
2. Formal Methods (FM).

[5] https://taxfix.de/

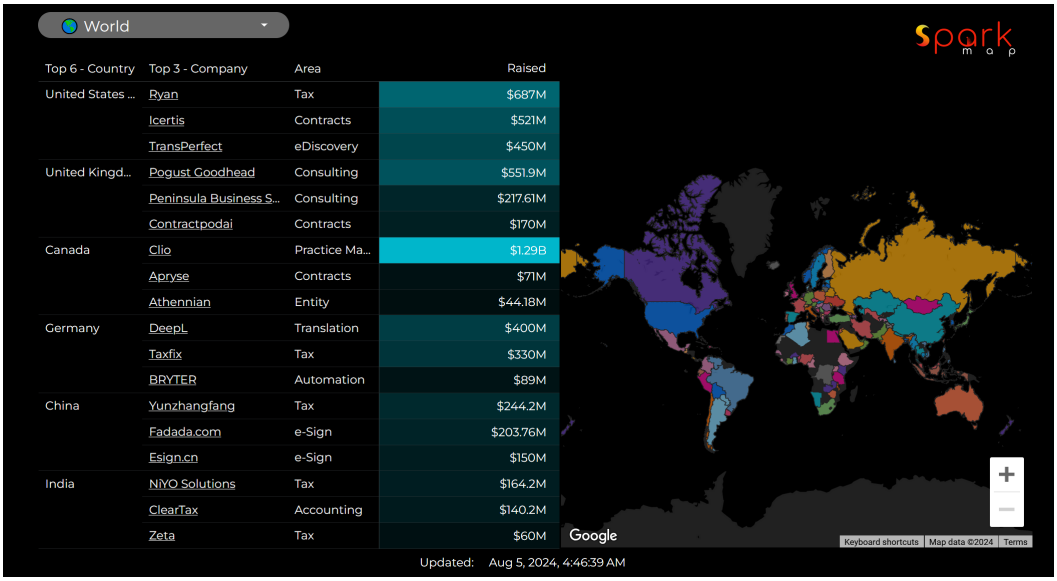


Figure 1.1: Total funding raised by startups in e-law and digital normative systems. Source: `https://www.legalcomplex.com/map/`.

Limitations of NLP approaches. While accessible, statistical approaches suffer from reliability issues. According to the startup Truth Systems[6], as presented on April 2024 [7], the accuracy of ChatGPT-4 when used for legal reasoning was approximately 59.2%. This lack of precision is attributed to hallucinations in LLMs. Figure 1.2 presents an example of such hallucinations, where a prompt regarding four individuals (Lucas, Amelia, Benjamin, and Olivia) suing each other is analyzed. The response is evaluated using a hallucination detector tool: red highlights indicate proven hallucinations; green highlights indicate verified facts and arguments; and orange highlights indicate unverified facts due to tool limitations. The query concerns a complex jurisdictional matter regarding the U.S. Federal Court's ability to accept cases involving plaintiffs from different states with a specific claim amount. The model's inability to strictly adhere to jurisdictional logic underscores the risks of relying solely on probabilistic models for compliance.

The tool relies on *semi-formal* techniques to identify hallucinations. In particular, it applies lightweight programmatic checks, such as named-entity detection, to ensure that no additional names are introduced between the prompt and the generated response. While effective for spotting certain surface-level inconsistencies, this approach does not constitute a complete formal reasoning and cannot guarantee logical correctness of the underlying legal analysis.

In the scope of this dissertation, we are interested in fully formal methods.

[6] `https://www.truthsystems.ai/`
[7] `https://law.stanford.edu/2024/04/02/youre-invited-codex-meeting-on-april-4-truth-systems-standard-draft/`

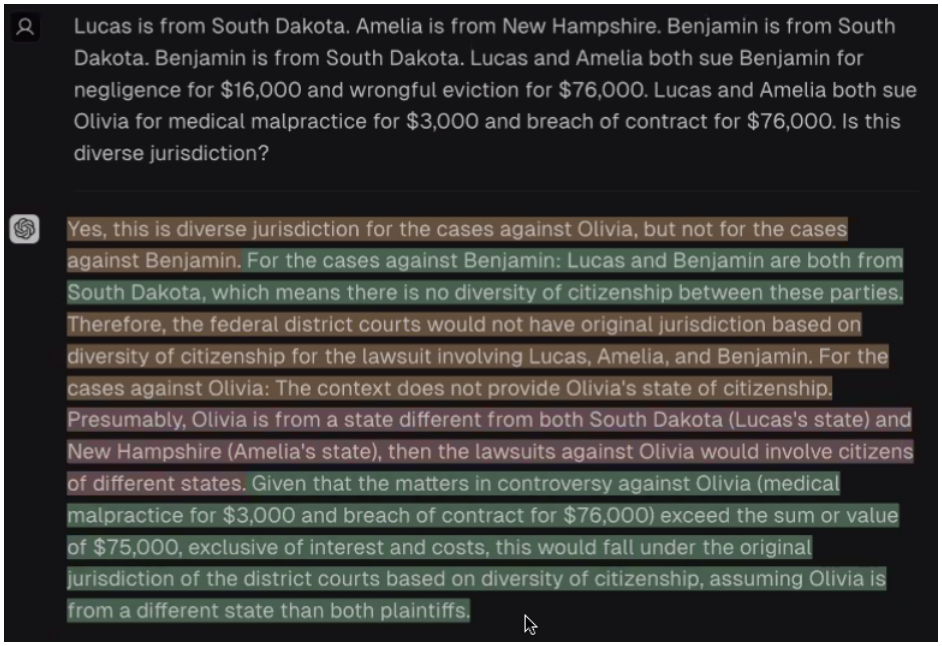


Figure 1.2: Example of hallucination in ChatGPT 4 regarding normative reasoning.

1.3 What are formal methods?

Formal methods rest on a simple but demanding principle. System descriptions and their expected behavior are treated as mathematical objects, and questions about their behavior are formulated as precise logical or algebraic problems. Rather than relying on informal reasoning about expected behavior, one commits to an explicit syntax, a formal semantics, and a notion of correctness that admits proof or algorithmic analysis. This shift increases the modeling effort, but it replaces persuasive arguments with verifiable outcomes such as theorems, counterexamples, or computed verdicts.

At their core, formal methods enforce a clear separation between concerns that are often entangled in informal design. First, they fix a representation of behavior, for example, traces, transition systems, automata, or logical structures. Second, they specify properties of interest, typically expressed as logical formulas, contracts, or invariants. Third, they define mathematically grounded procedures that relate behavior to properties, such as satisfaction, violation, refinement, or equivalence. This separation is what enables scalability, compositional reasoning, and reuse across domains.

Formal methods can be broadly classified by the nature of their inputs and the type of analysis they provide; we just refer to some of them:

Model-based verification. Model checking and automata-theoretic techniques operate on explicit or symbolic models of system behavior, such as finite-state machines, timed automata, or transition systems. Properties are specified in temporal or modal logics. The analysis is algorithmic and typically exhaustive. Either the model satisfies the property, or a concrete counterexample is produced. These methods are particularly effective for control systems, communication protocols, and reactive components, where behavior can be finitely represented or soundly abstracted [CES09, BK08].

Deductive and proof-based methods. Theorem-proving and proof-carrying approaches reason at the level of logical theories rather than at the level of state exploration. Systems are described by axioms, program contracts, inference rules, fix-points, or invariants, and correctness is established by constructing proofs in a formal calculus. These methods naturally handle infinite-state systems and rich data domains, but they depend more strongly on human guidance and appropriate abstraction choices [Hoa93, Dij76, CC77].

Specification and system-contract methods. Specification languages, design-by-contract techniques, and normative formalisms focus on what a system must provide. The primary objects of analysis are system contracts with formal semantics, depending on the system's domain of application [Inc22, BCN+18].

Runtime and monitor-based methods. Runtime verification and monitoring techniques analyze executions as they are observed. Instead of establishing correctness for all possible behaviors, they evaluate concrete traces against a formal specification. The result may be a verdict that evolves over time, such as satisfaction, violation, or an inconclusive status. These methods sacrifice completeness for practicality and are effective when full system models are unavailable or when systems operate in open or partially observable environments [LS09, BFFR18].

Across all these categories, the unifying feature is not a particular logic or algorithm, but a commitment to explicit semantics and mechanically checkable reasoning. Formal methods do not remove complexity. They make it explicit, structure it, and delimit precisely what can and cannot be guaranteed. This is why they become indispensable once informal reasoning ceases to be reliable, as is the case in safety-critical, legally regulated, or norm-governed systems.

1.4 Applications of Formal Methods in Computational Law

The intersection of legal theory and computer science has consolidated into the discipline of *Computational Law*. This field concerns the mechanization of legal reasoning and the representation of laws, regulations, and contracts as executable code.

Within this discipline, the application of formal methods aims to bridge the structural gap between expressive natural languages, which are rich but inherently ambiguous, and automated analysis techniques that must remain decidable and precise. By relying on mathematical logic, formal methods provide computer-processable representations of normative systems that support unambiguous interpretation, systematic verification, and reproducible reasoning. This enables compliance analysis under realistic constraints, including explicit time bounds, concurrent actions, and distributed agency. Current applications of this approach span several domains:

Automating Compliance and Requirements. Formal methods enable computational systems to determine exactly which legal requirements apply to a specific situation. This includes determining obligations, prohibitions, and permissions, as well as detecting violations. Systems can automatically check if a set of facts violates a regulation and identify whether compensatory clauses (remedies) have been satisfied [GMS06].

Business Process and Software Engineering. Formal methods integrate legal compliance into the design phase of technical systems. Organizations can verify if business process blueprints comply with regulations (such as Anti-Money Laundering laws) before deployment [Aal11].

Regulatory Analysis and Design. Regulators use formal tools to improve the quality of legislation. Digital versions of legislation allow agencies to identify inconsistencies, logical loops, or redundant conditions [BTV06].

Safety and Autonomous Systems. In real-time systems, formal methods encode complex rules for machine execution. For example, researchers have encoded overtaking clauses from Queensland road rules into logic to test how autonomous vehicles interpret safe distances and "clear views" in simulation [BGB+20].

1.5 Scope and Contributions of the Dissertation

While the general application of formal methods to law is promising, significant challenges remain in handling the dynamic aspects of normative systems. This dissertation focuses specifically on the challenges of *timed and distributed ought-to-do normative systems*.

Existing approaches often treat norms as static constraints or verify them centrally. However, real-world contracts and regulations involve multiple agents operating asynchronously, where the timing of actions (e.g., deadlines, durations) is as critical as the actions themselves.

Furthermore, in distributed settings, agents may have different or conflicting understandings of their normative state.

Contributions. This dissertation advances the formal analysis of normative systems under a deliberately limited, practitioner-oriented notion of expressivity. Rather than aiming for maximal logical generality, the focus is on fragments that reflect how norms are actually formulated, interpreted, and enforced in regulatory and contractual practice, where time bounds, deadlines, and role-specific duties dominate, and where reasoning must remain decidable and operational.

To the best of our knowledge, the main contributions are the following:

Con1 We introduce the first logical framework that treats *timed normative conflicts* as explicit semantic objects. The framework provides a precise notion of conflict between time-constrained obligations and prohibitions, a quantitative measure of the degree of conflict within a normative specification, and an automatic conflict-elimination procedure that preserves semantic equivalence whenever repair is possible. This moves normative analysis beyond binary consistency checks, enabling graded reasoning about normative incoherence.

Con2 We develop the first algorithmic account of responsibility/blame attribution for non-compliance in the setting of two agents collaborating under a contract. The proposed logic and monitoring procedure distinguish between failure and lack of *attempt*, allowing responsibility to be assigned even when collaborative actions fail due to delays, interference, or partial execution. This distinction, central in normative practice, is made formally precise and operationally verifiable.

Together, these contributions demonstrate that restricting expressivity in a principled manner enables stronger forms of automated reasoning about normative systems, while remaining closely aligned with the conceptual models used in legal and regulatory practice.

1.6 Structure of the Dissertation

This thesis is structured into the following three chapters:

Chapter 2 Introduces the basic data structures, logical concepts, and notation necessary for the understanding of this dissertation.

Chapter 3 Presents the static analysis of normative specifications under metric time and studies the formal notion of timed normative conflicts.

Chapter 4 Addresses the monitoring of normative interactions between two agents in the context of contracts, focusing on collaboration and blame assignment.

Chapter 5 The results of the overall work are summed up and discussed again.

Additionally, directions for future research are identified.

2 Preliminaries

This chapter introduces the mathematical structures that underlie the temporal component of our normative framework, together with the basic models used to represent system executions. We focus on representations that allow timed constraints to be stated, combined, and analyzed symbolically, without committing to a particular execution or monitoring model. In parallel, we introduce trace-based word models that serve as the semantic carriers for observed behavior and interaction. The material in this chapter is intentionally elementary and self contained. Its purpose is not to introduce new logical operators, but to establish precise domains for manipulating time ranges and executions algebraically, which will later be embedded into normative specifications and monitoring constructions.

2.1 Interval Set over Integers

Timed constraints in normative systems are rarely punctual. Obligations and prohibitions typically apply over contiguous or fragmented time ranges, and reasoning directly over individual time points leads to unnecessary blow up. For this reason, we adopt sets of intervals as our basic representation of temporal constraints. Interval sets provide a compact and canonical way to capture possibly discontinuous time ranges, while still supporting precise arithmetic operations such as intersection, union, and difference. Before introducing interval sets and their algebra, we first fix the underlying numerical domain on which these intervals are defined.

2.1.1 Natural Numbers and Extended Natural Numbers

Let $\mathbb{N}$ denote the set of all positive integers including zero, i.e. $\mathbb{N} := \{0,1,2,3,\dots\}$ and let *less* ($<$) and *less or equal* ($\leq$) denote the usual order relations on $\mathbb{N}$. We extend $\mathbb{N}$ by adding the element ∞, denoting infinity, forming the set of extended natural numbers: $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$. We define the order relations for $\mathbb{N}^\infty$: $<_\infty$ and $\leq_\infty$:

$$\text{For } n,m \in \mathbb{N}^\infty, n <_\infty m \text{ if and only if } \begin{cases} n \in \mathbb{N} \text{ and } m \in \mathbb{N} \text{ and } n < m, \\ n \in \mathbb{N} \text{ and } m = \infty. \end{cases}$$

$$\text{For } n,m \in \mathbb{N}^{\infty}, n \leq_{\infty} m \text{ if and only if } \begin{cases} n \in \mathbb{N} \text{ and } m \in \mathbb{N} \text{ and } n \leq m, \\ n \in \mathbb{N} \text{ and } m = \infty, \\ n = \infty \text{ and } m = \infty. \end{cases}$$

2.1.2 Intervals

Definition 1 (Interval)**.** *A* non-empty interval *is a pair* $[t_{min}, t_{max}]$ *with* $t_{min} \in \mathbb{N}$, $t_{max} \in \mathbb{N}^{\infty}$, *and* $t_{min} \leq_{\infty} t_{max}$. *The* empty interval, *written* $\varnothing$, *contains no elements. The set of all valid intervals is defined as* $\mathbb{I} := \{[t_{min}, t_{max}] \mid t_{min} \leq_{\infty} t_{max}\} \cup \{\varnothing\}$. *For* $n \in \mathbb{N}$, *we write* $n \in [t_{min}, t_{max}]$ *iff* $t_{min} \leq_{\infty} n \leq_{\infty} t_{max}$.

Informally, a non-empty interval $[t_{min}, t_{max}]$ *represents* a contiguous set of numbers, that is: $\{n \mid n \in [t_{min}, t_{max}]\}$, and the empty interval $\varnothing$ represents the empty set of natural numbers. The *size* of an interval I, written $|I|$, relates to the cardinality of the set that it represents. Thus, the size for $\varnothing$ is zero and for valid non-empty intervals:

$$|[a,b]| := \begin{cases} b - a - 1 & \text{if } b \in \mathbb{N}, \\ \infty & \text{if } b = \infty. \end{cases}$$

Example 1. *Consider the following examples of intervals:*

- *If* $t_{min} := 2$ *and* $t_{max} := 5$, *then* $I := [2,5]$ *represents* $\{2,3,4,5\}$.
- *If* $t_{min} := 3$ *and* $t_{max} := \infty$, *then* $I := [3,\infty]$ *represents* $\{t \in \mathbb{N} \mid t \geq 3\}$.
- *If* $t_{min} := 3$ *and* $t_{max} := 3$, *then* $I := [3,3]$ *represents* $\{3\}$.
- *If* $t_{min} := 3$ *and* $t_{max} := 0$, *then* $I := [3,0]$ *is not a valid interval.*

We transform the timed constraints from the use case to intervals:

1. *Let* $I_1 := [9,16]$ *represent the timed constraint between 9:00 and 16:00 from the Delivery Clause (Table Table 3.1).*
2. *Let* $I_2 := [8,24]$ *represent the constraint between 8:00 and 24 from the Maintenance Announcement (Table Table 3.2).*
3. *Let* $I_3 := [10,14]$ *represent the constraint between 10:00 and 14:00 from the Safety regulation (Table Table 3.1).*

2.1.3 Allen's Algebra and Unifiability

Allen's algebra [All83] consists of 13 exhaustive and mutually exclusive relations between *abstract*[1] pairs of intervals. To provide an intuition, in Figure 2.1 we give a visual interpretation of these relations. The different cases are based on the relative comparison of the start and end points of the intervals. In Table Table 2.1 we specify the constraints associated with each case for the domain of intervals $\mathbb{I}$ defined on natural numbers.

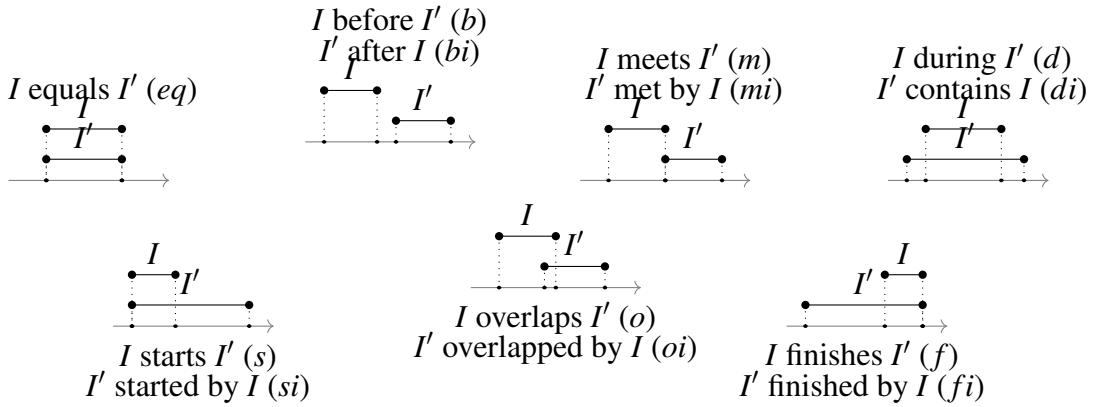


Figure 2.1: Visual representation of all base relations from Allen's Interval Algebra (continuous Intervals)

We additionally define the *non-unifiable* relation $\overline{U}$: the relation captures that the union of two intervals is not closed under $\mathbb{I}$. This relation is a special case of the *before* relation:

$$\overline{U}([a,b],[c,d]) \text{ iff } ([a,b+1] \text{ before } [c,d]) \text{ or } ([c,d+1] \text{ before } [a,b]).$$

Example 2. *We show below examples illustrating some relations between intervals:*

1. ***Before**: $I := [0,3]$ and $I' := [4,7]$.*
   I is before I', since I ends before I' starts.
2. ***Non-unifiable**: $I := [0,2]$ and $I' := [4,7]$.*
   I is not unifiable with I', since $[0,3]$ precedes $[4,7]$.
3. ***After**: $I := [5,8]$ and $I' := [0,4]$.*
   I is after I', since I starts after I' ends.
4. ***Meets**: $I := [0,3]$ and $I' := [3,6]$.*
   I meets I', since I ends exactly when I' starts.

[1] Not inherently tied to a specific numerical domain.

Notation	Interpretation	Necessary constraints
$\boldsymbol{I < I'}$ $\boldsymbol{I' > I}$	I precedes I' I' precedes I	$\max(I) <_\infty \min(I')$
$\boldsymbol{I = I'}$	I is equal to I' I' is equal to I	$\min(I) = \min(I')$, $\max(I) = \max(I')$
$\boldsymbol{I\ m\ I'}$ $\boldsymbol{I'\ mi\ I}$	I meets I' I' meets I	$\max(I) = \min(I')$
$\boldsymbol{I\ o\ I'}$ $\boldsymbol{I'\ oi\ I}$	I overlaps with I' I' overlaps with I	$\min(I) <_\infty \min(I') < \max(I) <_\infty \max(I')$
$\boldsymbol{I\ s\ I'}$ $\boldsymbol{I'\ si\ I}$	I starts I' I' starts I	$\min(I) = \min(I')$, $\max(I) <_\infty \max(I')$
$\boldsymbol{I\ d\ I'}$ $\boldsymbol{I'\ di\ I}$	I during I' I' during I	$\min(I') <_\infty \min(I)$, $\max(I) <_\infty \max(I')$
$\boldsymbol{I\ f\ I'}$ $\boldsymbol{I'\ fi\ I}$	I finishes I' I' finishes I	$\max(I) = \max(I')$, $\min(I) >_\infty \min(I')$

Table 2.1: Constraint on intervals from $\mathbb{I}$ to satisfy a relation from Allen's Algebra

5. ***Overlaps***: $I := [0,4]$ *and* $I' := [3,6]$.
   I overlaps I', since I starts before I' and ends after I' starts.
6. ***Starts***: $I := [2,5]$ *and* $I' := [2,8]$.
   I starts I', since I and I' have the same start time, but I ends before I'.
7. ***Finishes***: $I := [4,7]$ *and* $I' := [1,7]$.
   I finishes I', since I and I' have the same end time, but I starts after I'.
8. ***During***: $I := [3,6]$ *and* $I' := [1,8]$.
   I is during I', since I starts after I' starts and ends before I' ends.
9. ***Contains***: $I := [0,7]$ *and* $I' := [2,5]$.
   I contains I', since I starts before I' and ends after I' ends.
10. ***Non-unifiable***: $I := [0,3]$ *and* $I' := [5,\infty]$.
    $\overline{U}(I,I')$, *since* $3+1 < 5$.

The intersection operator: $\cap : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ is defined as follows:

$$[a,b] \cap [c,d] := \begin{cases} \varnothing & \text{iff} \quad [a,b] \text{ before } [c,d] \text{ or } [c,d] \text{ before } [a,b] \\ [max(a,c), min(b,d)] & \text{otherwise} \end{cases}$$

The operators we define now are partially defined as the interval domain not being closed under these operations.

The *union* operator $\cup : \mathbb{I} \times \mathbb{I} \rightharpoonup \mathbb{I}$ is partially defined as follows:

$$[a,b] \cup [c,d] := \begin{cases} \text{undefined} & \text{iff } \overline{U}([a,b],[c,d]), \\ [min(a,b,c,d), max(a,b,c,d)] & \text{otherwise.} \end{cases}$$

The partial interval difference operator $\ominus : \mathbb{I} \times \mathbb{I} \rightharpoonup \mathbb{I}$ is the non-commutative, partially defined operation removing all elements from the interval I' from the interval I in $I \ominus I'$:

$$[a,b] \ominus [c,d] := \begin{cases} \text{undefined} & \text{iff } [c,d] \text{ during } [a,b], \\ [a,b] & \text{iff } [a,b] \cap [c,d] = \emptyset, \\ \varnothing & \text{iff } [a,b] \subseteq [c,d], \\ [a,c-1] & \text{iff } [a,b] \text{ meets or overlaps or finished by } [c,d], \\ [d+1,b] & \text{iff } [c,d] \text{ meets or overlaps or starts by } [a,b]. \end{cases}$$

Example 3. *Let us consider some examples concerning the operator* $\ominus$*:*

- $[2,8] \ominus [4,6] = \textit{undefined}$ *(during)*
- $[0,3] \ominus [5,7] = [0,3]$ *(empty intersection)*
- $[2,5] \ominus [1,6] = \varnothing$ *(subset)*
- $[1,5] \ominus [3,7] = [1,2]$ *(overlaps on left)*
- $[0,5] \ominus [3,\infty] = [0,2]$ *(overlaps left with infinity)*
- $[4,8] \ominus [1,5] = [6,8]$ *(overlaps on right)*
- $[3,\infty] \ominus [0,5] = [6,\infty]$ *(overlaps right with infinity)*

We use now the difference operator on the three-time intervals from our use case: $I_1 \ominus I_2 = [9,16] \ominus [12,\infty] = [9,11]$*. On the other hand,* $I_1 \ominus I_3 = [9,16] \ominus [10,14]$ *is undefined as* $[10,14]$ *during* $[9,16]$*.*

2.1.4 Interval Sets

As demonstrated in the previous subsection, the interval domain $\mathbb{I}$ is not closed under union and difference operations. To address this limitation, we introduce the concept of interval sets. An *interval set*, denoted as $\mathfrak{I}$, is a finite set of intervals equipped with a condition to ensure minimal representation. This condition is called the *canonical form*, which ensures that the interval set contains the smallest number of non-overlapping and non-adjacent intervals necessary to represent a given set of time points.

Definition 2 (Canonical Interval Set)**.** *An* interval set $\mathfrak{I}$ *is said to be in* canonical form *if no two intervals in* $\mathfrak{I}$ *are unifiable. Formally:*

$$\forall\{I, I'\} \subseteq \mathfrak{I} : \overline{U}(I, I').$$

where $\overline{U}(I, I')$ *denotes that* I *and* I' *are not unifiable (i.e.,* $I \cup I'$ *is not an interval from* $\mathbb{I}$*).*

For the remainder of this paper, we adopt the following notational convention: we use $\mathfrak{I}^*$ *to refer to interval sets composed exclusively of finite (bounded) intervals. In contrast, we use* $\mathfrak{I}$ *to denote interval sets that may contain infinite intervals, such as those extending to* $+\infty$*.*

The size of a set of intervals $\mathfrak{I}$*, written* $|\mathfrak{I}|$*, is defined as the sum of the sizes of the individual intervals it contains:*

$$|\mathfrak{I}| := \sum_{I \in \mathfrak{I}} |I|.$$

The cardinality *of the* $\mathfrak{I} := \{I_0, I_1, \ldots, I_n\}$*, written* $\mathsf{card}(\mathfrak{I})$*, is the number of element intervals in the set,* $|\mathfrak{I}| := n + 1$*.*

Example 4 (Canonical form for interval sets)**.** *Consider the interval set* $\mathfrak{I}' := \{[0,2],[3,6]\}$*. The two intervals forming this interval set are unifiable as* $[0,2] \cup [3,6] = [0,6]$*. The canonical representation is thus* $\mathfrak{I} := \{[0,6]\}$*.*

Consider the interval set $\mathfrak{I} := \{[0,2],[5,7],[10,\infty]\}$*.*
The pair $[0,2]$ *and* $[5,7]$ *satisfy* $\overline{U}([0,2],[5,7])$*.*
Similarly, $[5,7]$ *and* $[10,\infty]$ *satisfy* $\overline{U}([5,7],[10,\infty])$*.*
Finally, $[0,2]$ *and* $[10,\infty]$ *also satisfy* $\overline{U}([0,2],[10,\infty])$*.*

These representations enable a symbolic encoding of timed constraints, such as those described in our use case. For instance, the obligations from MA, as shown in Table Table 3.1, that an action is permitted *between 02:00 and 04:00, and between 08:00 and 24:00*—can be expressed as the canonical interval set:

$$\mathfrak{I}_3 := \{[2,4],[8,24]\}.$$

Similarly, constraints from DC and SR can each be represented using singleton interval sets. The constraint *between 09:00 and 16:00*, for example, corresponds to:

$$\mathfrak{I}_1 := \{[9,16]\},$$

while *between 10:00 and 14:00* is represented as:

$$\mathfrak{I}_2 := \{[10,14]\}.$$

We adopt this interval-set representation as a uniform basis for the symbolic manipulation of temporal constraints within the logical framework developed in the following sections.

Definition 3 (Expansion Factor of an Interval Set)**.** *Let $\mathfrak{I}$ be a finite set of intervals. The* expansion factor *$\iota(\mathfrak{I})$ of $\mathfrak{I}$ is defined as the ratio between the total number of discrete points covered by the intervals in $\mathfrak{I}$ and the number of intervals in $\mathfrak{I}$, i.e.,*

$$\iota(\mathfrak{I}) := \frac{|\mathfrak{I}|}{card(\mathfrak{I})} \quad ,$$

where $|\mathfrak{I}|$ is the sum of the sizes (number of points) of the intervals in $\mathfrak{I}$, and $\mathrm{card}(\mathfrak{I})$ is the number of intervals in $\mathfrak{I}$.

Example 5 (Minimal Expansion Factor)**.** *Consider the interval set*

$$\mathfrak{I}_1 = \{[3,3],[5,5],[10,10]\}.$$

The cardinality is $\mathrm{card}(\mathfrak{I}_1) = 3$, and the size is

$$|\mathfrak{I}_1| = (3-3+1)+(5-5+1)+(10-10+1) = 1+1+1 = 3.$$

Therefore, the expansion factor is

$$\iota(\mathfrak{I}_1) = \frac{3}{3} = 1,$$

Example 6 (Large Expansion Factor)**.** *Consider now the interval set*

$$\mathfrak{I}_2 = \{[1,1000],[1050,1050],[2000,2000]\},$$

which consists of one large interval and two punctual intervals. Here,

$$card(\mathfrak{I}_2) = 3,$$

and

$$|\mathfrak{I}_2| = (1000-1+1)+(1050-1050+1)+(2000-2000+1) = 1000+1+1 = 1002.$$

Thus, the expansion factor is

$$\iota(\mathfrak{I}_2) = \frac{1002}{3} = 334,$$

Showing a large blow-up, mainly caused by the large intervals.

2.1.5 Arithmetic of Interval Sets

Sub-element An interval I is a *sub-element* of $\mathfrak{I}$ (written $I \Subset \mathfrak{I}$) if it is included in one of the elements of the set of intervals. We write: $I \Subset \mathfrak{I}$ iff $\exists I' \in \mathfrak{I} : I \subseteq I'$. We abuse the notation for time points $t \in \mathbb{N}$ and write $t \Subset \mathfrak{I}$ iff $[t,t] \Subset \mathfrak{I}$.

Inclusion We say $\mathfrak{I}$ is *included* in $\mathfrak{I}'$ and write: $\mathfrak{I} \subset \mathfrak{I}'$ iff $\forall I \in \mathfrak{I}\ \exists I' \in \mathfrak{I}' : I \subset I'$.

Example 7. *Given* $\mathfrak{I} := \{[1,4],[6,9]\}$*:*

- *The interval* $[2,3] \Subset \mathfrak{I}$ *because* $[2,3] \subseteq [1,4] \in \mathfrak{I}$.
- *The time point* $7 \Subset \mathfrak{I}$ *because* $[7,7] \subseteq [6,9] \in \mathfrak{I}$.

Let $\mathfrak{I}' = \{[0,4],[6,\infty]\}$*. Then* $\mathfrak{I} \subseteq \mathfrak{I}'$ *since*

- $[1,4] \subseteq [0,4] \in \mathfrak{I}'$,
- $[6,9] \subseteq [6,\infty] \in \mathfrak{I}'$.

For the sets of intervals $\mathfrak{I}_1 := \{[9,16]\}$ *and* $\mathfrak{I}_3 := \{[2,4],[8,24]\}$ *from our use case we have:* $\mathfrak{I}_1 \subset \mathfrak{I}_3$.

Intersection The intersection of two sets of intervals $\mathfrak{I}$ and $\mathfrak{I}'$ is the set of intervals containing all the intervals present in both interval sets. The operator is commutative and is defined as:

$$\mathfrak{I} \cap \mathfrak{I}' := \begin{cases} \emptyset & \text{if } \forall I \in \mathfrak{I}, \forall I' \in \mathfrak{I}' : I \cap I' = \varnothing, \\ \{I \cap I' \mid I \in \mathfrak{I}, I' \in \mathfrak{I}', I \cap I' \neq \varnothing\} & \text{otherwise.} \end{cases}$$

Example 8. *Consider the following examples:*

- *If* $\mathfrak{I} := \{[1,5],[8,10],[19,\infty]\}$ *and* $\mathfrak{I}' := \{[3,7],[9,12]\}$*, we have* $\mathfrak{I} \cap \mathfrak{I}' = \{[3,5],[9,10]\}$*, as* $[1,5] \cap [3,7] = [3,5]$ *and* $[8,10] \cap [9,12] = [9,10]$.
- *If* $\mathfrak{I} := \{[1,5],[8,10]\}$ *and* $\mathfrak{I}' := \{[6,7],[12,\infty]\}$ *then* $\mathfrak{I} \cap \mathfrak{I}' = \emptyset$ *(as none of the intervals from* $\mathfrak{I}$ *intersect with any interval from* $\mathfrak{I}'$*).*

For the intervals $\mathfrak{I}_1$*,* $\mathfrak{I}_2$*, and* $\mathfrak{I}_3$ *related respectively to* DC*,* SR*, and* MA *from the use case:*

- $\mathfrak{I}_1 \cap \mathfrak{I}_2 = \{[9,16]\} \cap \{[10,14]\} = \{[10,14]\}$.
- $\mathfrak{I}_1 \cap \mathfrak{I}_3 = \{[9,16]\} \cap \{[2,4],[8,24]\} = \{[9,16]\}$.

Union

The union of two interval sets $\mathfrak{I}$ and $\mathfrak{I}'$ is recursively computed by the successive union of each interval $I_i \in \mathfrak{I}'$ with $\mathfrak{I}$:

$$\mathfrak{I} \cup \mathfrak{I}' := \left\{ ((\mathfrak{I} \cup I_1) \cdots) \cup I_n) \;\middle|\; I_1, \ldots, I_n \in \mathfrak{I}' \right\}.$$

The union of an interval set $\mathfrak{I}$ with a single interval I is given by:

$$\mathfrak{I} \cup I := \left\{ \left(\bigcup_{I_i \in U(I,\mathfrak{I})} (I \cup I_i) \right) \right\} \circ \overline{U}(I, \mathfrak{I}),$$

where $\circ$ denotes set concatenation, and the subset of unifiable and non unifiable interval with I from $\mathfrak{I}$ are defined as:

$$\overline{U}(I, \mathfrak{I}) := \{ I_i \in \mathfrak{I} \mid \overline{U}(I, I_i) \},$$
$$U(I, \mathfrak{I}) := \{ I_i \in \mathfrak{I} \mid I_i \notin \overline{U}(I, \mathfrak{I}) \}.$$

Example 9. *Let* $\mathfrak{I} := \{[1,5],[9,11]\}$ *and* $\mathfrak{I}' := \{[3,7],[9,12]\}$*:*

$$\begin{aligned}
\mathfrak{I} \cup \mathfrak{I}' &= \big(\{[1,5],[9,11]\} \cup [1,7]\big) \cup [9,12] && \textit{(unrolling)} \\
&= \big(\{[1,5] \cup [3,7]\} \circ \{[9,11]\}\big) \cup [9,12] && (\textit{as } \overline{U}([9,11],[3,7])) \\
&= \{[1,7],[9,11]\} \cup [9,12] && \textit{(simplification)} \\
&= \{[9,11] \cup [9,12]\} \circ \{[1,7]\} && (\textit{as } \overline{U}([9,12],[1,7])) \\
&= \{[1,7],[9,12]\}
\end{aligned}$$

Difference (Reduction)

The *difference* of an interval set $\mathfrak{I}$ by an interval set $\mathfrak{I}'$, denoted $\mathfrak{I} \ominus \mathfrak{I}'$, is the set of intervals in $\mathfrak{I}$ excluding those that overlap with $\mathfrak{I}'$:

$$\mathfrak{I} \ominus \mathfrak{I}' := \bigcup_{I_i \in \mathfrak{I}} (I \ominus \mathfrak{I}') \text{ with } I \ominus \mathfrak{I}' = \bigcap_{I_i \in \mathfrak{I}'} (I \ominus I_i).$$

Where the *redefined* reduction of two intervals is now computed as *set of intervals* instead of intervals as defined in the arithmetic of intervals: $\ominus : \mathbb{I} \times \mathbb{I} \to 2^{\mathbb{I}}$ is defined as :

$$[a,b] \ominus [c,d] := \begin{cases} \{[a,c-1],[d+1,b]\} & \text{iff } [c,d] \text{ during } [a,b], \\ \{[a,b]\} & \text{iff } [a,b] \cap [c,d] = \varnothing, \\ \emptyset & \text{iff } [a,b] \subseteq [c,d], \\ \{[a,c-1]\} & \text{iff } [a,b] \text{ meets or overlaps or finished by} [c,d], \\ \{[d+1,b]\} & \text{iff } [c,d] \text{ meets or overlaps or starts by} [a,b]. \end{cases}$$

Remark 1. *An alternative definition could be* $I \ominus I' := \{I_i \mid I_i \subseteq I \text{ and } I_i \not\subseteq I'\}$.

Example 10. *Let* $\mathfrak{I} = \{[1,10]\}$ *and* $\mathfrak{I}' = \{[3,5],\ [7,8]\}$*:*

- $[1,10] \ominus [3,5] = \{[1,2],[6,10]\}$.
- $[1,10] \ominus [7,8] = \{[1,6],[9,10]\}$.

Thus, $\mathfrak{I} \ominus \mathfrak{I}' = \{[1,2],\ [6,6],\ [9,10]\}$.

From the intervals $\mathfrak{I}_1, \mathfrak{I}_2, \mathfrak{I}_3$ related to our three normative systems from the use case, we have:

- $\mathfrak{I}_1 \ominus \mathfrak{I}_2 = \{[9,16]\} \ominus \{[10,14]\} = \{[9,9],[15,16]\}$.
- $\mathfrak{I}_1 \ominus \mathfrak{I}_3 = \{[9,16]\} \ominus \{[2,4],[8,24]\} = \emptyset$.

2.2 Discrete-Action Word Models

Since our normative specifications constrain what agents ought to do, we take discrete actions as primitive and represent executions as discrete-action words. In contrast to control-theoretic and physical models based on continuous signals or state trajectories, our setting is action-centric: we record which actions occur, and we reason about their order and, when needed, their timing. Accordingly, a word is a sequence of discrete events that captures "what happened", rather than a valuation that describes "what holds" over time. These words form the basic semantic objects for the logics developed in subsequent chapters. When we speak of traces, we mean recorded words obtained from concrete executions.

We begin by establishing three word models depending on their timing settings and the canonical morphisms between them. Subsequently, we study how to synchronize two words of the same type.

2.2.1 Notations on Discrete-Action Words

For the remainder of this dissertation, let Σ denote a non-empty set of discrete *actions*. Before introducing the specific timing models, we fix a small algebraic core shared by all discrete-action words, specifically the basic sequence structure and standard operators.

Events and Words. An *event*, denoted by e, is an element of an event domain $\mathbb{E}$. The specific structure of $\mathbb{E}$ depends on the timing model derived from Σ (e.g., $\mathbb{E} = \Sigma$ for untimed words). We denote the set of *finite words* over $\mathbb{E}$ by $\mathbb{E}^*$, the set of *infinite words* by $\mathbb{E}^\omega$, and the set of all words by $\mathbb{E}^\infty := \mathbb{E}^* \cup \mathbb{E}^\omega$.

The empty word, denoted by $\langle - \rangle$, is the unique finite word containing no events. The set of non-empty finite words is Σ^+, we have $\Sigma^* = \langle - \rangle \cup \Sigma^+$.

Notation for non-empty finite words. Let $w = \langle e_1, e_2, \ldots, e_n \rangle \in \mathbb{E}^+$ be a non-empty finite word. We adopt *1-based indexing* where indices correspond to the position of events in the sequence.

- *Size*: The length of the word denoted by $|w|$, is defined as $|w| := n$.
- *Positions*: The set of valid positions is $\mathsf{pos}(w) := \{1, 2, \ldots, n\}$.
- *Indexing*: For any $k \in \mathsf{pos}(w)$, the event at position k is $w[k] := e_k$.
- *Prefix*: For any $1 \leq k \leq n$, the prefix of length k is denoted by w_k:

  $$w_k := \langle e_1, \ldots, e_k \rangle.$$

  By convention, we set up: $w_0 = \langle - \rangle$ and $w_n = w$.
- *Suffix*: For any $0 \leq k \leq n$, the suffix remaining *after* the first k events is denoted by w^k:

  $$w^k := \langle e_{k+1}, \ldots, e_n \rangle.$$

  Note that $w^0 = w$ and $w^n = \langle - \rangle$.

Words vs. Traces. We conceptually distinguish a *finite word* (conceptual model) from a *finite trace* (an observed behavior of a system/agent). However, formally, both are finite sequences of events in $\mathbb{E}^*$. In this dissertation, we use "word" as the default technical term and "trace" when emphasizing that the sequence is the result of a recorded observation.

Omega notation. For any non-empty finite word $u \in \mathbb{E}^+$, written $u = \langle e_1, \ldots, e_n \rangle$ with $n \geq 1$, we write u^ω to denote the infinite word obtained by repeating u indefinitely:

$$u^\omega := \langle e_1, \ldots, e_n, e_{n+1}, \ldots, e_{n+n}, \ldots \rangle \in \mathbb{E}^\omega.$$

where for $k > n$, the event e_k is defined by the modulo relation:

$$e_k := e_{((k-1) \bmod n)+1},$$

so that u^ω cycles through the events of u indefinitely.

Prefix of an infinite word. Let $v \in \mathbb{E}^\omega$ be an infinite word and let $u \in \mathbb{E}^*$ be a finite word. We say that u is a *prefix* of v, written $u \preceq v$, if there exists $k \in \mathbb{N}$ such that $|u| = k$ and

$$\forall i \in \{1, \ldots, k\}, \quad u[i] = v[i].$$

Prefix order. For any finite word $u \in \mathbb{E}^*$ and any word $v \in \mathbb{E}^\infty$, we write $u \preceq v$ if $u = v_k$ for some $k \in \mathbb{N}$. If additionally $u \neq v$, we write $u \prec v$.

Generic Projection. For a subset of events $S \subseteq E$, the projection $w{\restriction}S$ is the subsequence obtained by deleting all events e_k such that $e_k \notin S$.

2.2.2 Logical-Time Words

Logical-time words capture *order only*, abstracting away any notion of specific timing. They are the staple semantic objects in interleaving models of process calculi (such as CCS[Mil89] and CSP[Hoa78]). Moreover, in Mazurkiewicz trace theory, one starts from strings over an alphabet and then forms *traces* by identifying strings that differ only by swapping adjacent independent actions (that is, a trace is an equivalence class of words modulo an independence relation) [DR95].

Definition 4 (Logical-time word, notation τ^{lt})**.** *A (finite) logical-time word over an alphabet Σ is a sequence:*

$$\tau^{\mathrm{lt}} = \langle (a_1), (a_2), \ldots, (a_n) \rangle \in \Sigma^*,$$

where $a_k \in \Sigma$ for all $k \in \{1, \ldots, n\}$.

We now introduce basic operators on logical-time words. We start with concatenation, then define a counting function for action occurrences. Both notions will be reused when relating logical-time words to the richer time models introduced next.

Definition 5 (Concatenation of logical-time words)**.** *Let* $\tau_1^{\mathrm{lt}} = \langle (a_1), \ldots, (a_n) \rangle \in \Sigma^*$ *and* $\tau_2^{\mathrm{lt}} = \langle (b_1), \ldots, (b_m) \rangle \in \Sigma^*$*. Their* concatenation*, written* $\tau_1^{\mathrm{lt}} \circ \tau_2^{\mathrm{lt}}$*, is the logical-time word in* Σ^* *defined by*

$$\tau_1^{\mathrm{lt}} \circ \tau_2^{\mathrm{lt}} \;:=\; \langle (a_1), \ldots, (a_n), (b_1), \ldots, (b_m) \rangle.$$

$$\tau_1^{\mathrm{lt}} \circ \langle - \rangle \;:=\; \tau_1^{\mathrm{lt}}.$$
$$\langle - \rangle \circ \tau_2^{\mathrm{lt}} \;:=\; \tau_2^{\mathrm{lt}}.$$

Additionally, the following holds for the concatenation of any two logical-time words:

- *Length*: $|\tau_1^{\mathrm{lt}} \circ \tau_2^{\mathrm{lt}}| = |\tau_1^{\mathrm{lt}}| + |\tau_2^{\mathrm{lt}}|$.
- *Indexing*: for $k \in \mathsf{pos}(\tau_1^{\mathrm{lt}} \circ \tau_2^{\mathrm{lt}})$,

$$(\tau_1^{\mathrm{lt}} \circ \tau_2^{\mathrm{lt}})[k] := \begin{cases} \tau_1^{\mathrm{lt}}[k], & \text{if } 1 \le k \le |\tau_1^{\mathrm{lt}}|, \\ \tau_2^{\mathrm{lt}}[k - |\tau_1^{\mathrm{lt}}|], & \text{if } |\tau_1^{\mathrm{lt}}| < k \le |\tau_1^{\mathrm{lt}}| + |\tau_2^{\mathrm{lt}}|. \end{cases}$$

The next operator abstracts from positions and records how often a given action occurs in a word.

Definition 6 (Counting function on logical-time words)**.** *Let* $a \in \Sigma$ *and let* $\tau^{\mathrm{lt}} = \langle (a_1), \ldots, (a_n) \rangle \in \Sigma^*$ *be a logical-time word. The* count *of* a *in* τ^{lt}*, written* $\#(a, \tau^{\mathrm{lt}})$*, is defined by*

$$\#(a, \tau^{\mathrm{lt}}) \;:=\; \big|\{\, k \in \mathsf{pos}(\tau^{\mathrm{lt}}) \mid \tau^{\mathrm{lt}}[k] = a \,\}\big|.$$

Example 11 (Logical-time operators)**.** *Let* $\Sigma = \{\mathit{pick}, \mathit{handover}\}$*. Consider the word:*

$$\tau^{\mathrm{lt}} = \langle (\mathit{pick}), (\mathit{handover}) \rangle.$$

Size and Indexing: $|\tau^{\mathrm{lt}}| = 2$, $\tau^{\mathrm{lt}}[1] = \mathit{pick}$.
Counts: $\#(\mathit{handover}, \tau^{\mathrm{lt}}) = 1$.
Projection: *with* $A = \{\mathit{handover}\}$, $\tau^{\mathrm{lt}} \restriction A = \langle (\mathit{handover}) \rangle$.
Concatenation: $\tau^{\mathrm{lt}} \circ \langle (\mathit{pick}) \rangle = \langle (\mathit{pick}), (\mathit{handover}), (\mathit{pick}) \rangle$.

2.2.3 Metric-Timed Words

Context. Metric-time words extend the previous model by recording *which* action occurs and *when* it occurs. They are defined in research works on automata with time [AD94].

Definition 7 (Finite metric-time words of actions)**.** *A* finite metric-time word τ *is defined as a finite sequence of time events,* $\tau \in (\Sigma \times \mathbb{N})^*$.

$$\tau := \langle (a_1,t_1),(a_2,t_2),\ \ldots,\ (a_n,t_n)\rangle$$

with $a_k \in \Sigma$*,* $t_k \in \mathbb{N}$*, and strictly increasing timestamps* $0 \le t_1 < \cdots < t_n$.

Definition 8 (Auxiliary operators on metric-time words)**.** *Let* $\tau^{\mathrm{mt}} = \langle (a_1,t_1),\ldots,(a_n,t_n)\rangle \in (\Sigma \times \mathbb{N})^*$.

- *The* label *and* timestamp *projections of the kth event are defined by*

$$\mathsf{lab}(\tau^{\mathrm{mt}},k) := a_k, \qquad \mathsf{ts}(\tau^{\mathrm{mt}},k) := t_k \qquad (k \in \mathsf{pos}(\tau^{\mathrm{mt}})).$$

- *The* time set *of* τ^{mt} *is*

$$\mathsf{time}(\tau^{\mathrm{mt}}) := \{\, t_k \mid k \in \mathsf{pos}(\tau^{\mathrm{mt}}) \,\}.$$

- *The* last time point *is defined by*

$$\mathsf{last}(\tau^{\mathrm{mt}}) := \begin{cases} t_n, & \text{if } n > 0, \\ 0, & \text{if } n = 0. \end{cases}$$

- *The* time span *is*

$$\mathsf{span}(\tau^{\mathrm{mt}}) := \begin{cases} t_n - t_1, & \text{if } n > 0, \\ 0, & \text{if } n = 0. \end{cases}$$

- *For* $a \in \Sigma$*, the* count *of* a *in* τ^{mt} *is*

$$\#(a,\tau^{\mathrm{mt}}) := \big|\{\, k \in \mathsf{pos}(\tau^{\mathrm{mt}}) \mid \mathsf{lab}(\tau^{\mathrm{mt}},k) = a \,\}\big|.$$

We denote the set of all finite metric-timed words by $\mathbb{T} := (\Sigma \times \mathbb{N})^*$. Unlike the logical-time setting, where events are compared solely by order, this model supports quantitative reasoning about the precise timing between events. This added expressiveness, however, restricts algebraic composition: concatenation is not defined for arbitrary pairs of words, as the combined sequence must respect the strictly increasing order of timestamps. To formalize these timing properties, the next definitions introduce two auxiliary operators: Δ measures the elapsed time between positions, while ρ retrieves the action occurring at a specific absolute time.

Definition 9 (Timed distance between two events)**.** *Let* $\tau^{\mathrm{mt}} = \langle (a_1,t_1),\ldots,(a_n,t_n)\rangle \in (\Sigma \times \mathbb{N})^*$ *be a metric-time word. For any two indices* $i, j \in \mathsf{pos}(\tau^{\mathrm{mt}})$ *with* $i < j$*, the* time distance *between event* i *and event* j *is*

$$\Delta(\tau^{\mathrm{mt}},(i,j)) \;:=\; t_j - t_i.$$

Definition 10 (Action Lookup Function)**.** *The* action lookup function $\rho : \mathbb{T} \times \mathbb{N} \to \Sigma \cup \{\mathsf{undefined}\}$ *returns the action performed at absolute time* t *in a metric-time word; if no event occurs at time* t*, it returns the special symbol* undefined*:*

$$\rho\big(\langle (a_1,t_1),\ldots,(a_n,t_n)\rangle, t\big) := \begin{cases} a_i & \text{if } t_i = t \text{ for some } i \in \{1,\ldots,n\}, \\ \mathsf{undefined} & \text{otherwise.} \end{cases}$$

Basic operators and short hands. For $A \subseteq \Sigma$, projection $\tau^{\mathrm{mt}}{\restriction}A \in (A \times \mathbb{N})^*$ keeps exactly the pairs (a,t) where $a \in A$.
Prefixes can be defined by *length* $\tau^{\mathrm{mt}}[1,k]$, or by *time* $\tau^{\mathrm{mt}}{\restriction}\{t \le T\}$ (shorthand: $\tau^{\mathrm{mt}}\{t \le T\}$).

Example 12 (Metric-timeo perators)**.** *Consider an alphabet* $\Sigma = \{\mathit{badge\_tapped}, \mathit{door\_open}\}$.

$$\tau^{\mathrm{mt}} = \langle (\mathit{badge\_tapped},1),(\mathit{door\_open},3)\rangle.$$

Size and Indexing: $|\tau^{\mathrm{mt}}| = 2$, $\tau^{\mathrm{mt}}[1] = (\mathit{badge\_tapped},1)$.
Lookup action: $\rho(\tau^{\mathrm{mt}},1) = \mathit{badge\_tapped}$ *and* $\rho(\tau^{\mathrm{mt}},2) = \mathsf{undefined}$.
Time properties: $\mathsf{time}(\tau^{\mathrm{mt}}) = \{1,3\}$, $\mathsf{span}(\tau^{\mathrm{mt}}) = 2$.
Time-prefix: $\tau^{\mathrm{mt}}\{t \le 2\} = \langle (\mathit{badge\_tapped},1)\rangle$.
Timed distance*: the distance between event* 1 *and event* 2 *is* $\Delta(\tau^{\mathrm{mt}},(1,2)) = 3-1 = 2$.

Metric-time words provide *precise* quantitative information, in particular the exact time distance between any two events in the same word, captured by differences of their timestamps. For this model, concatenation is not defined for arbitrary pairs of words: if two words both contain an event at the same time point, then concatenating them would violate the requirement of strictly increasing timestamps. Hence, concatenation is only defined when the timestamp sets are disjoint and the last time point of the first word is strictly smaller than the first time point of the second word.

2.2.4 Synchronous-Time Words

Synchronous-time words represent behavior under a single global logical clock. Each tick corresponds to one round of computation, and inactivity is made explicit via the stutter symbol "$-$". This model underlies synchronous languages such as Esterel[BC84], Lustre[Hal93], and LoLa[dSS+05].

The synchronous hypothesis. The synchronous hypothesis fixes the intended abstraction level: within one clock cycle, the system reads its inputs, computes the next state, and emits its outputs within the same logical instant. Computation is therefore treated as taking zero time, and time advances only between cycles. Accordingly, behavior is naturally represented as a word indexed by $\mathbb{N}$, where each position corresponds to one tick and records the global observation at that tick, for instance as a global state or as a set of events. Related synchronous process-algebra variants also exist, most notably Milner's synchronous CCS (SCCS), which adopts the same global-clock view and interprets behavior as lockstep rounds of actions [Mil83].

Definition 11 (Synchronous-time word, notation τ^{st})**.** *Fix an alphabet Σ and a stutter symbol "$-$" with $- \notin \Sigma$. A (finite)* synchronous-time word *is a finite sequence of events over $(\Sigma \cup \{-\})$*

$$\tau^{\mathrm{st}} \;=\; \langle (s_1), (s_2), \ldots, (s_T) \rangle \;\in\; (\Sigma \cup \{-\})^*,$$

where $s_t \in \Sigma \cup \{-\}$ represents the observation at round t and T is called the horizon.

Synchronous-time words sit between logical-time words and metric-time words. Like logical time, they are discrete sequences, but they come with an explicit global round index that plays the role of a coarse clock. Like metric time, one can talk about *when* something happens, but only in terms of round numbers rather than absolute timestamps. The crucial difference to both earlier models is that the *absence* of an action is itself observable, represented by the stutter symbol "$-$".

Basic operators and short hands. For $A \subseteq \Sigma$:

- *Stutter-preserving projection* $\tau^{\mathrm{st}}{\restriction}A \in (A \cup \{-\})^*$ keeps letters in A and all "$-$".
- *Logical (stutter-erasing) projection* $\mathsf{erase}_{-}(\tau^{\mathrm{st}}{\restriction}A) \in A^*$ drops all "$-$".

For $a \in \Sigma$, the *count* of a in τ^{st} is $\#(a, \tau^{\mathrm{st}}) := \big|\{ t \in \mathsf{pos}(\tau^{\mathrm{st}}) \mid \tau^{\mathrm{st}}[t] = a \}\big|$, and the set of *active rounds* is $\mathsf{act}(\tau^{\mathrm{st}}) := \{ t \in \mathsf{pos}(\tau^{\mathrm{st}}) \mid \tau^{\mathrm{st}}[t] \neq - \}$.

Example 13 (controller at 1 Hz, illustrating all operators)**.** *Consider HVAC (Heating, Ventilation, and Air Conditioning) actuator (1) and a security Sensor (2), both controlled by a synchronous controller:*

$$\Sigma_1 = \{\mathit{heat\_on}\}, \qquad \Sigma_2 = \{\mathit{door\_lock}\}.$$

In one episode, we record

$$\tau_1^{\mathrm{st}} = \langle (-), (-), (\mathit{heat\_on}) \rangle, \qquad \tau_2^{\mathrm{st}} = \langle (-), (\mathit{door\_lock}), (-) \rangle.$$

Size, positions, indexing:
$|\tau_1^{\mathrm{st}}| = |\tau_2^{\mathrm{st}}| = 3$ *with horizon* $T = 3$, $\mathrm{pos}(\tau_1^{\mathrm{st}}) = \{1,2,3\}$, $\tau_1^{\mathrm{st}}[3] = \mathit{heat\_on}$.

Counts and active rounds:
$\#(\mathit{heat\_on}, \tau_1^{\mathrm{st}}) = 1$, $\mathrm{act}(\tau_1^{\mathrm{st}}) = \{3\}$.

Stutter-preserving projection:
with $A = \emptyset$, $\tau_1^{\mathrm{st}} \restriction A = \langle(-),(-),(-)\rangle$.

Logical (stutter-erasing) projection:
$\mathrm{erase}_{-}(\tau_1^{\mathrm{st}} \restriction A) = \langle - \rangle$, *while* $\mathrm{erase}_{-}(\tau_1^{\mathrm{st}}) = \langle(\mathit{heat\_on})\rangle$.

Round-prefix:
$\tau_2^{\mathrm{st}}[1,2] = \langle(-),(\mathit{door\_lock})\rangle$.

2.2.5 Connecting the Three Action Word Models

The objective of this subsection is to connect the three action-centric word models introduced above. Given a metric-time word τ^{mt}, we introduce two canonical morphisms that deliberately discard part of the information: the *logical-time projection* LT, which erases time stamps while preserving the order of events, and the *synchronous padding* ST, which aligns events to a global round clock by inserting explicit stutter symbols "$-$". We set up the clock frequency of the synchronous-time word to correspond to a single unit of metric time.

Summary diagram. Figure 2.2 summarizes the relationship between the three word models using a running toy example with two events. Starting from a metric-time word, synchronous padding ST aligns events to a global round clock by inserting explicit stutters "$-$", and logical projection LT forgets timing information and retains only the event order. The diagram also makes explicit that the co-domain changes across the mappings: ST moves from time event pairs to round-indexed letters, whereas LT yields a pure order-only word.

From metric time to logical time. The map LT drops time stamps but preserves labels and their order. It is the coarsest view that still distinguishes different event sequences.

Lemma 1 (Logical-time projection LT: order-only view)**.** *Let* $\tau^{\mathrm{mt}} = \langle(a_1,t_1),\ldots,(a_n,t_n)\rangle$ *be a time word. There exists a unique logical word* $\tau^{\mathrm{lt}} = \mathsf{LT}(\tau^{\mathrm{mt}}) \in \Sigma^*$ *satisfying:*

- ***Number of event preservation****:* $|\tau^{\mathrm{lt}}| = |\tau^{\mathrm{mt}}| = n$.
- ***Action occurrence preservation****:* $\forall a \in \Sigma, \#(a,\tau^{\mathrm{lt}}) = \#(a,\tau^{\mathrm{mt}})$.

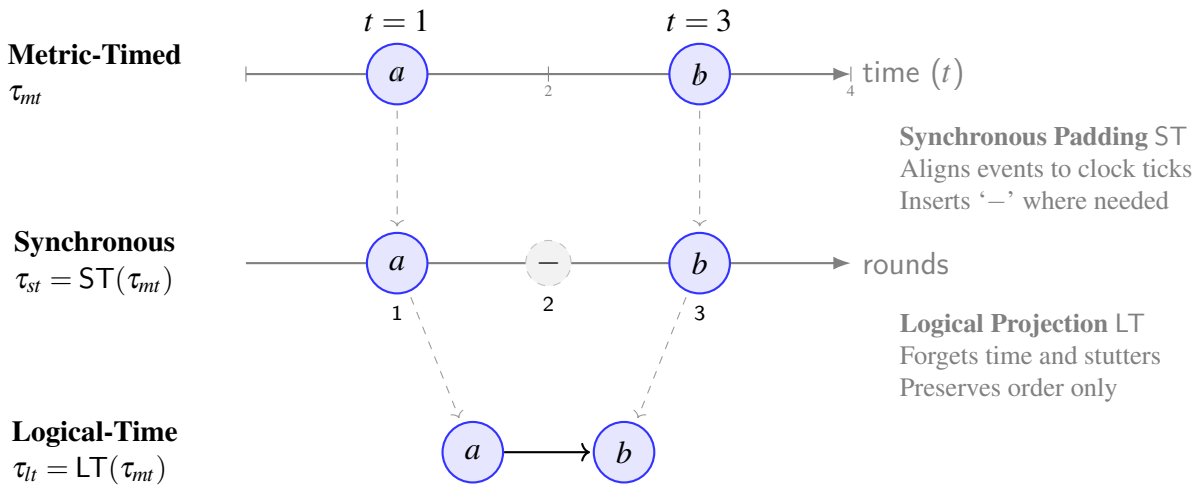


Figure 2.2: Relationship between metric-time, synchronous-time, and logical-time word models, and the forgetful morphisms ST and LT on a running example.

- ***Event order preservation***: *If* $\tau^{\mathrm{mt}}[k]$ *precedes* $\tau^{\mathrm{mt}}[\ell]$, *then* $\tau^{\mathrm{lt}}[k]$ *precedes* $\tau^{\mathrm{lt}}[\ell]$ *in* τ^{lt}.
- ***Projection compatibility***: $\forall A \subseteq \Sigma, (\mathsf{LT}(\tau^{\mathrm{mt}}))\restriction A = \mathsf{LT}(\tau^{\mathrm{mt}}\restriction A)$.
- ***Timing is forgotten***: LT *is invariant under strictly increasing retiming of timestamps.*

Proof. Construction. Define $\mathsf{LT}(\tau^{\mathrm{mt}}) := \langle (a_1), (a_2), \ldots, (a_n) \rangle$. All properties follow from erasing time stamps while preserving labels and their order. □

From metric time to synchronous time. The map ST aligns events to a global round clock by inserting "−" at rounds with no event; erasing "−" brings us back to logical time.

Lemma 2 (Synchronous padding ST: round-indexed view)**.** *Let* τ^{mt} *be a metric-time word. There exists a unique synchronous word* $\tau^{\mathrm{st}} = \mathsf{ST}(\tau^{\mathrm{mt}}) \in (\Sigma \cup \{-\})^*$ *such that, writing* $T := \mathsf{last}(\tau^{\mathrm{mt}})$ *(with* $T = 0$ *if* $\tau^{\mathrm{mt}} = \langle - \rangle$*):*

- ***Size / horizon relation***: *If* $|\tau^{\mathrm{mt}}| > 0$, *then* $|\mathsf{ST}(\tau^{\mathrm{mt}})| = T + 1$ *and* $\mathsf{pos}(\mathsf{ST}(\tau^{\mathrm{mt}})) = \{1, \ldots, T+1\}$.
- ***Exact time-of-occurrence***: *For any timestamp* $t \in \{0, \ldots, T\}$, *the synchronous event at index* $t+1$ *is determined by:*

$$\mathsf{ST}(\tau^{\mathrm{mt}})[t+1] = \begin{cases} \mathsf{lab}(\tau^{\mathrm{mt}}, k) & \text{if } \exists k : \mathsf{ts}(\tau^{\mathrm{mt}}, k) = t, \\ - & \text{otherwise.} \end{cases}$$

- ***Active rounds:*** $\mathsf{act}(\mathsf{ST}(\tau^{\mathrm{mt}})) = \{\, t+1 \mid t \in \mathsf{time}(\tau^{\mathrm{mt}}) \,\}$.
- ***Preservation of occurrences:*** $\forall a \in \Sigma, \#(a, \mathsf{ST}(\tau^{\mathrm{mt}})) = \#(a, \tau^{\mathrm{mt}})$.
- ***Equivalence to logical time:*** $\mathsf{erase}_{-}(\mathsf{ST}(\tau^{\mathrm{mt}})) = \mathsf{LT}(\tau^{\mathrm{mt}})$.

Proof. Construction. If $|\tau^{\mathrm{mt}}| = 0$, set $\mathsf{ST}(\tau^{\mathrm{mt}}) := \langle - \rangle$. Otherwise, let $T := \mathsf{last}(\tau^{\mathrm{mt}})$. We construct the word of length $T+1$. For every integer $t \in \{0, \ldots, T\}$ (representing the metric time), we define the event at index $t+1$ as:

$$\mathsf{ST}(\tau^{\mathrm{mt}})[t+1] := \begin{cases} a_k & \text{if } \exists k \text{ s.t. } t_k = t, \\ - & \text{otherwise.} \end{cases}$$

Since metric timestamps are strictly increasing, there is at most one k for any t, making this well-defined. All properties follow from this construction. □

On non-invertibility. Both LT and ST are *many-to-one*. In particular, LT is *not* invertible. One cannot reconstruct a unique metric or synchronous word from logical time alone. Conversely, given $\mathsf{ST}(\tau^{\mathrm{mt}})$ under the one-tick-per-unit assumption, the timestamps are recoverable from the indices of non-stutter symbols, up to the chosen time unit and the choice of time origin.

Takeaway. LT and ST are the canonical forgetful maps from metric time to, respectively, order-only and round-synchronous views. They preserve exactly the event properties listed above and discard the rest.

2.2.6 Synchronization of Words

Design rationale. Having defined words for generic systems, we now turn to the interaction between systems. Different communities fix different *time assumptions*, which determine how local words are aligned. Process calculi (CCS/CSP) adopt *no global clock* and reason over order only, yielding *pure interleaving* and optionally *shared action handshakes*. Synchronous languages assume a *single global round clock* (lockstep). Timed automata use *absolute timestamps* and synchronize at equal times on shared actions.

In this section, we instantiate three operators consistent with our models for two specific agents, denoted by indices 1 and 2, with alphabets Σ_1 and Σ_2. We cover: (i) *asynchrony* on logical-time words (τ^{lt}), (ii) *lockstep synchrony* on synchronous words (τ^{st}), and (iii) *synchronous handshake* on synchronous words with an explicit set A of shared actions.

Asynchrony Operator on Logical Time

The asynchronous operator models purely interleaved joint behavior, generating *all shuffles* that preserve each agent's local order. No global clock or rendezvous is assumed. To avoid accidental identification of simultaneous events, the global alphabet is taken as the *disjoint union* $\Sigma = \Sigma_1 \uplus \Sigma_2$, tagging each action with its agent of origin. This is the standard interleaving semantics of CCS and CSP.

Definition 12 (Asynchronous interleaving on τ^{lt})**.** *Let* $\tau_1^{\mathrm{lt}} \in \Sigma_1^*$ *and* $\tau_2^{\mathrm{lt}} \in \Sigma_2^*$ *be logical-time words. The* asynchronous interleaving operator $\|\!| : \Sigma_1^* \times \Sigma_2^* \longrightarrow 2^{(\Sigma_1 \uplus \Sigma_2)^*}$ *is defined recursively by:*

$$\tau_1^{\mathrm{lt}} \|\!| \tau_2^{\mathrm{lt}} = \begin{cases} \{\, \tau_2^{\mathrm{lt}} \,\}, & \tau_1^{\mathrm{lt}} = \langle - \rangle, \\ \{\, \tau_1^{\mathrm{lt}} \,\}, & \tau_2^{\mathrm{lt}} = \langle - \rangle, \\ \{\, a \cdot w \mid w \in (u \|\!| b \cdot v) \,\} \cup \{\, b \cdot w \mid w \in (a \cdot u \|\!| v) \,\}, & \tau_1^{\mathrm{lt}} = a \cdot u,\ \tau_2^{\mathrm{lt}} = b \cdot v. \end{cases}$$

Where $a \in \Sigma_1$, $b \in \Sigma_2$, $u \in \Sigma_1^*$ *and* $v \in \Sigma_2^*$.

As a result, we have that every $w \in \tau_1^{\mathrm{lt}} \|\!| \tau_2^{\mathrm{lt}}$ satisfies the projection property: $w{\restriction}\Sigma_1 = \tau_1^{\mathrm{lt}}$ and $w{\restriction}\Sigma_2 = \tau_2^{\mathrm{lt}}$.

Example 14 (Warehouse pick/scan, pure interleaving)**.** *Let* $\Sigma_1 = \{\mathit{pick}\}$, $\Sigma_2 = \{\mathit{scan}\}$. *System 1:* $\tau_1^{\mathrm{lt}} = \langle(\mathit{pick}), (\mathit{pick})\rangle$. *System 2:* $\tau_2^{\mathrm{lt}} = \langle(\mathit{scan})\rangle$. *Then* $\tau_1^{\mathrm{lt}} \|\!| \tau_2^{\mathrm{lt}}$ *contains three words:* $\langle(\mathit{scan}), (\mathit{pick}), (\mathit{pick})\rangle$, $\langle(\mathit{pick}), (\mathit{scan}), (\mathit{pick})\rangle$, *and* $\langle(\mathit{pick}), (\mathit{pick}), (\mathit{scan})\rangle$.

Synchronous (Lockstep) Operator on Synchronous Time

The *lockstep* operator combines two synchronous words by aligning them round by round under a shared global clock. If the two words have different horizons, the shorter one is right-padded with the stutter symbol "$-$" so that both align on the common index set $\{1, \ldots, T\}$, where $T = \max(T_1, T_2)$.

Definition 13 (Lockstep zip on τ^{st})**.** *Let* $\tau_1^{\mathrm{st}} = \langle(s_1), \ldots, (s_{T_1})\rangle$ *and* $\tau_2^{\mathrm{st}} = \langle(r_1), \ldots, (r_{T_2})\rangle$ *be two synchronous words. Extend the shorter word with stutters "$-$" up to* $T := \max(T_1, T_2)$. *The* lockstep zip *operator is defined by:*

$$\tau_1^{\mathrm{st}} \,\|_{\mathrm{lock}}\, \tau_2^{\mathrm{st}} \;:=\; \big\langle (s_1, r_1),\ (s_2, r_2),\ \ldots,\ (s_T, r_T) \big\rangle.$$

Each position $t \in \{1, \ldots, T\}$ *of the result records the simultaneous round of both agents.*

The result is a synchronous word over the product alphabet $(\Sigma_1 \cup \{-\}) \times (\Sigma_2 \cup \{-\})$.

Synchronous Operator With Handshake Actions

In many systems, certain actions can only be executed *jointly* and must occur in *the same round* for both agents. Let $A \subseteq \Sigma_1 \cap \Sigma_2$ denote the set of *handshake actions*. The idea is that if one agent performs a handshake action $a \in A$ at some round, then the other agent must also perform a at that exact round. This principle, specified in synchronous CCS [Mil83], requires handshake actions to be treated as simultaneous, mutually synchronized events.

Definition 14 (Lockstep with handshakes on synchronous words)**.** *Let $A \subseteq \Sigma_1 \cap \Sigma_2$ be a nonempty set of handshake actions. The lockstep with handshakes operator, written $\tau_1^{\mathrm{st}} \parallel_{\mathrm{hs}}^{A} \tau_2^{\mathrm{st}}$, takes two synchronous words τ_1^{st} and τ_2^{st}, extends them to the same length T, and applies the handshake constraint.*

For the operator to be defined, the following condition must be true for every round $t \in \{1,\dots,T\}$ and every $a \in A$:

$$(s_t = a \ \vee \ r_t = a) \ \Rightarrow \ (s_t = r_t = a).$$

If this condition is met, the result is a sequence of pairs from the product alphabet:

$$\tau_1^{\mathrm{st}} \parallel_{\mathrm{hs}}^{A} \tau_2^{\mathrm{st}} \ = \ \langle (s_1, r_1), (s_2, r_2), \dots, (s_T, r_T) \rangle.$$

If the condition is not met, the operator is undefined, which means a deadlock or an invalid execution.

The definition above gives a word over the raw product alphabet $(\Sigma_1 \cup \{-\}) \times (\Sigma_2 \cup \{-\})$. However, it is often helpful to combine joint actions into single events to get a standard word structure. To do this, we need a richer alphabet that can show shared actions, private actions, and private actions happening at the same time.

Definition 15 (Collapsed lockstep with handshakes)**.** *Let $P_1 = \Sigma_1 \setminus A$ and $P_2 = \Sigma_2 \setminus A$ be the sets of private actions. The collapsed alphabet $\Sigma_{\parallel}$ is defined as:*

$$\Sigma_{\parallel} \ := \ A \ \cup \ P_1^{(1)} \ \cup \ P_2^{(2)} \ \cup \ (P_1^{(1)} \times P_2^{(2)}),$$

Here, $P_i^{(i)}$ means the private actions are labeled with the agent's ID.

The collapsed lockstep with handshakes is found by applying the function coll_A to each pair (s_t, r_t) in the raw lockstep word:

$$\mathsf{coll}_A(s_t, r_t) = \begin{cases} a, & \text{if } s_t = r_t = a \in A, \\ s_t^{(1)}, & \text{if } s_t \in P_1, \ r_t = -, \\ r_t^{(2)}, & \text{if } r_t \in P_2, \ s_t = -, \\ (s_t^{(1)}, r_t^{(2)}), & \text{if } s_t \in P_1, \ r_t \in P_2, \\ -, & \text{if } s_t = r_t = -. \end{cases}$$

Example 15 (Handover as handshake, failures and successes)**.** *Let* $\Sigma_1 = \{\mathit{pick}, \mathit{handover}\}$, $\Sigma_2 = \{\mathit{scan}, \mathit{handover}\}$, *and* $A = \{\mathit{handover}\}$. *The private sets are* $P_1 = \{\mathit{pick}\}$ *and* $P_2 = \{\mathit{scan}\}$.

- Success (aligned handshake). $\tau_1^{\mathrm{st}} = \langle(\mathit{pick}), -, \mathit{handover}\rangle$, $\tau_2^{\mathrm{st}} = \langle -, \mathit{scan}, \mathit{handover}\rangle$. *The collapse gives* $\langle(\mathit{pick}^{(1)}),\ (\mathit{scan}^{(2)}),\ (\mathit{handover})\rangle$.
- Failure (mismatched handshake). $\tau_1^{\mathrm{st}} = \langle -, \mathit{handover}, -\rangle$, $\tau_2^{\mathrm{st}} = \langle \mathit{handover}, -, -\rangle$. *The constraint fails at* $t = 1$ *when Agent 2 is ready but Agent 1 is not, and at* $t = 2$ *when the situation is reversed. In this case, the operator is undefined.*
- Private simultaneous actions (non-handshake). $\tau_1^{\mathrm{st}} = \langle -, \mathit{pick}, -\rangle$, $\tau_2^{\mathrm{st}} = \langle -, \mathit{scan}, -\rangle$. *At round* $t = 2$ *(metric time 1),* $s_2 = \mathit{pick}$ *is in* P_1 *and* $r_2 = \mathit{scan}$ *is in* P_2. *The collapsed result is* $\langle -,\ (\mathit{pick}^{(1)}, \mathit{scan}^{(2)}),\ -\rangle$.

To sum up, this section presented three models for discrete-action words and explained how they relate to each other, as summarized in Table 2.2. We started by setting up an abstract algebraic framework for both finite and infinite words. Then, we described three specific word types: (i) logical-time words, which only keep the order of actions, (ii) metric-time words, which capture better timing order using strictly increasing timestamps, and (iii) synchronous-time words, which organize observations by rounds and show inactivity through stuttering. We linked these models using the forgetful morphisms LT and ST, which let us view a metric-time execution in (i) terms of order only, or (ii) as round-synchronous. Next, we introduced three ways for two agents to synchronize: asynchronous interleaving $\|\!|$ for logical time, lockstep zip $\|_{\mathrm{lock}}$ for synchronous time, and lockstep with handshakes $\|^A_{\mathrm{hs}}$, plus a version that combines everything into a single word. These tools form the basis for interpreting multi-agent contracts and for defining how monitoring and verdicts work under different synchrony settings.

Table 2.2: Summary of discrete-action word models, morphisms, and synchronization operators.

Model / Concept	**Symbol**	**Domain**	**Key Characteristics**
Word Models			
Logical-Time	τ^{lt}	Σ^*	Order only; no timing info.
Metric-Timed	τ^{mt}	$(\Sigma \times \mathbb{N})^*$	Precise timestamps; quantitative timing.
Synchronous-Time	τ^{st}	$(\Sigma \cup \{-\})^*$	Round-based; explicit stutters $(-)$.
Canonical Morphisms			
Logical Projection	LT	$\tau^{\mathrm{mt}} \to \tau^{\mathrm{lt}}$	Forgets timestamps, preserves order.
Synchronous Padding	ST	$\tau^{\mathrm{mt}} \to \tau^{\mathrm{st}}$	Aligns events to rounds, inserts $-$.
Synchronization Operators			
Async. Interleaving	$\|\|\|$	$\tau^{\mathrm{lt}} \times \tau^{\mathrm{lt}}$	All order-preserving shuffles (interleaving).
Lockstep Zip	$\|\|_{\mathrm{lock}}$	$\tau^{\mathrm{st}} \times \tau^{\mathrm{st}}$	Aligns rounds strictly.
Handshake	$\|\|^{A}_{\mathrm{hs}}$	$\tau^{\mathrm{st}} \times \tau^{\mathrm{st}}$	Synchronizes shared A, interleaves private.

3 Reasoning about Metric-Timed Normative Conflicts

3.1 Overview

As discussed in the introduction chapter, normative systems provide the formal backbone for how organizations, individuals, software, and autonomous agents are expected to behave. A *normative specification* describes these expectations as a set of rules determining what an agent ought to do or what state of affairs ought to be. However, dealing with such specifications is inherently difficult. They combine different types of rules, such as obligations, prohibitions, and permissions, whose interplay can yield overlapping prescriptions. In many situations, complying with one norm causes the violation of another. This situation is referred to as a *normative conflict*. This chapter studies those conflicts in *ought-to-do* specifications with explicit temporal conditions.

3.1.1 Motivation

Norms are rarely timeless: they are activated, suspended, or expire under explicit temporal conditions. This temporal structure has motivated a long line of research in *temporal deontic logics* and *contract specification languages*. Early works such as [DK98, BDDM04] formalized temporal deontic constraints by linking obligations to deadlines. A second wave of studies [BB07] examined how unmet obligations propagate over time, capturing the principle that "what I fail to do today, I must do tomorrow," which formalizes the continuation of obligations after a missed deadline. In parallel, [GHRR07, GR11] introduced a temporal defeasible logic that characterizes different types of deadlines, offering a flexible framework for representing and reasoning about time-sensitive norms. This line of research enabled the modeling of norms that become active or expire at specific points in time, the idea that a rule is *in force* at a given instant due to a reparation clause following a violation or a triggering event. More recent frameworks, such as [HKZ12], C–O Diagrams [CPS14], and FCL [FH16], offer greater expressive power and aim to capture real-world normative dynamics.

Unlike in untimed settings, conflicts in timed specifications depend on *when* norms are active and *how many* are active at once. When several obligations share the same acti-

vation period, the agent may lack the capacity to plan and execute them all within that window. Conflicts also occur at *moments* when the agent is obliged to perform an action and at the same time prohibited from doing it. In short, time turns contradiction into a scheduling and capacity problem: we must reason about activation periods and point-wise clashes, not only about truth values.

Early approaches, such as Fenech et al. [FPS09], formalized conflict detection through reachability analysis over automata whose states were annotated with deontic labels: whenever a reachable state contained incompatible obligations or prohibitions, the contract was considered conflicting. While effective for identifying logical contradictions, this view abstracts away from when and how conflicts arise. As argued by Colombo Tosatto, Governatori, and Kelsen [CTGK14], such inconsistencies often result not from $O\alpha \wedge O\neg\alpha$ alone but from *temporal infeasibility*, where no timed trace can satisfy the conjunction of active norms. More recent work, such as N-Check [FMGY$^+$24], reduces the analysis of normative requirements to first-order satisfiability checking, diagnosing conflicts as unsatisfiable rule combinations.

Research gap and questions Although existing approaches successfully expose when a formula contains a conflict, they typically treat time symbolically and leave the precise definition of what *constitutes* a normative conflict open. This lack of formal precision raises three fundamental research questions:

RQ1 **Relation between Unsatisfiability and Conflict:** The relationship between general logical unsatisfiability and specific normative conflicts is currently not well formalized. Is every unsatisfiable formula a conflict? Conversely, does the presence of a conflict necessarily imply that the formula is unsatisfiable?

RQ2 **Conflict Quantification:** While satisfiability is intrinsically a binary property (true or false), can we define a quantitative measure for the *degree* of conflict within a specification?

RQ3 **Properties-Preserving Resolution:** Is it always possible to resolve conflicts in a formula without altering its inherent semantics or compromising its structural compactness?

3.1.2 Methodology

To address these challenges, we first formalize the potential behavior of agents as finite timed words by structuraly encoding it as a sequence of timed events formed by *a unique action* per event. The We introduce a light logic, μMTNL, allowing us to formalize normative specifications where time conditions are represented as finite interval sets. This trace-based semantics serves as the ground truth for our analysis.

We then narrow down the abstract notion of conflict into two fundamental types. The first is *deontic*, where rules contradict each other. The second is *ontic*, where rules demand more than the agent can physically perform. We show that these clashes often hide within complex time intervals. To expose them, we develop a rigorous transformation process that unfolds specifications into their atomic components. Crucially, we prove that this process is structure-preserving. It reveals strictly those conflicts present in the original design without creating spurious ones or erasing existing ones.

With these atomic conflicts exposed, we introduce a quantitative metric to measure the density of inconsistency in a system. We demonstrate that this measure is directly linked to whether a specification is logically solvable. A fully conflicting system is impossible to satisfy. Conversely, a partially conflicting one can always be automatically repaired. We provide an algorithm for this repair that effectively prunes the contradictory parts while preserving the valid compliant behaviors.

We then bridge the gap between this operational detection and theoretical logic. We prove that our defined conflict patterns are *enough to explain all causes* of the unsatisfiability of any formula. This ensures that our detection method is both sound and complete.

Finally, we address the practical trade-off between automation and conciseness. Eliminating conflicts automatically can sometimes result in verbose specifications. To mitigate this, we complement our algorithms with a *conflict calculus*. This set of inference rules allows for high-level symbolic reasoning. It enables designers to resolve conflicts manually and simplify specifications without needing to decompose the entire formula.

3.1.3 Contribution

In this chapter, we provide a logical framework for *metric-time normative conflicts* that deals with these issues in a mathematically precise manner. Our specific contributions, building up on our previous work [KLS21, KSL24] are as follows:

C1 **Formal Machinery:** We define a compact, action-based logic over finite interval sets. We provide algorithms for punctual and disjunctive normalization (*DPNF*) and a proof-theoretic calculus for conflict detection.

C1 **Conflict Characterization:** We offer precise definitions for conflicts and conditions for their correct detection. We prove that deontic and ontic conflicts correspond precisely to Minimal Unsatisfiable Subsets (MUS) in our logic. Conversely, we show that any MUS is necessarily an instance of one of these conflict types.

C1 **Quantitative Metrics:** We introduce the concept of *conflict density*, which measures the ratio of conflicting to total punctual obligations. We show that a conflict density of

1 implies unsatisfiability, while a density below 1 allows for a conflict-free, semantics-preserving pruning.

C1 **Elimination and Guarantees:** We provide sound and complete methods for equivalence-preserving conflict elimination. We analyze the verbosity induced by this process, demonstrating that while eliminating conflicts can increase the syntactic size of the specification, the blow-up remains bounded under well-defined structural conditions.

Altogether, this framework allows us to identify conflicts in a specification, explain their causes via interval arithmetic, measure their density, and eliminate them without altering the intended compliant behaviors.

3.2 Use Case: Goods Delivery Contract

Our use case examines a contract between a trucking company and a harbor, focusing on the daily delivery of potentially hazardous goods to the harbor. Table 3.1 presents a portion of the contract signed between the trucking company and the harbor. For clarity, we do not provide the full contractual agreement, but instead highlight select sections to illustrate our language and the occurrence of *potential conflicts*.

... The deliveries *shall* be made everyday to *either the gate terminal or the designated parking area of the harbor*. The delivery must be done *between 09:00 to 16:00* .	(DC)
... Delivery of dangerous goods to the harbor's parking lot is prohibited between 10:00 and 14:00.	(SR)
... When delivering goods, special restrictions may apply due to maintenance activity in the harbor.	(MC)
...	

Table 3.1: Contract between trucking company and a harbor company

Let us elaborate on the contract. There are individuals and legal entities involved, like the truck driver, the transportation company and the harbor.

The contract involves several clauses; first, a *Delivery Clause* (DC) for all goods. Moreover, there is a clause adding additional *Safety Regulations* (SR) when dangerous goods are involved. The third and final clause requires respecting any *Maintenance Clause* (MC) of the harbor when delivering goods.

Due to infrastructure damage at the harbor, the delivery of dangerous goods to the gate terminal is temporarily prohibited during the following hours: (1) Between 02:00 and 04:00, and (2) between 08:00 and 24:00.	(MA)

Table 3.2: Maintenance Announcement

The Delivery Clause (DC) specifies the designated delivery location and time frame (between 09:00 and 16:00) to be performed on a daily basis; the harbor carries on by loading the goods onto the ferry on fixed and reserved slots as specified in the contract, making the driver's delivery act mandatory in the delivery chain. The Safety Regulation clause SR prohibits the delivery of *dangerous* goods in the harbor's parking area between 10:00 and 14:00. The Maintenance Clause MC mandates compliance with announcements when delivering goods. These maintenance announcements are issued by the harbor's infrastructure management unit by providing a *Maintenance Announcement* document (MA) requiring consultation (cf. the document shown in Table 3.2). This document may be empty in case no announcement is in place or adds specific clauses.

Let us now reconsider the contract and perform a more conceptual analysis: Individuals and legal entities affected by the normative system are called *agents*. These agents are required to follow the prescriptions and rules of the contract or agreement, which are collectively termed *norms*. In this paper, we focus on formalizing norms concerning a *single* agent, leaving multi-agent scenarios for future work. Norms stipulate what the different agents are permitted, obliged or forbidden to do, as well as the penalties to be applied in case of non-compliance. As explained in the introduction, we focus here not on *full* normative systems but on a subset of normative statements.

An agent is typically not bound by just one norm but must comply with several norms simultaneously (all the norms in the given normative system(s) under consideration). The norms may be distributed over several documents. The norms are enacted (and enforced) at specific *time points*. In general, such time points are explicitly or implicitly written in the contract and they can be expressed in relative or absolute time. For instance, they can be relative to a certain date (e.g., when the contract was signed) or to the occurrence of certain events, or then expressed as dates, hours of a day, etc. Without loss of generality, we represent such time points as natural numbers.[1] More precisely, norms are associated with time *intervals*, and the agent must adhere to constraints across multiple such intervals. In the next section, we will make these notions more formal (e.g., we will define intervals and establish operations for *interval arithmetic*). Back to our use case, the driver's task is to deliver goods within specific time constraints defined as intervals as stipulated in the contract. Whenever the driver performs *actions* which are within the scope of the contract, we may consider that the contract is being *executed*. Such actions

[1] It is always possible to convert from relative to absolute time (and vice-versa) as well as to map hours and dates into natural numbers.

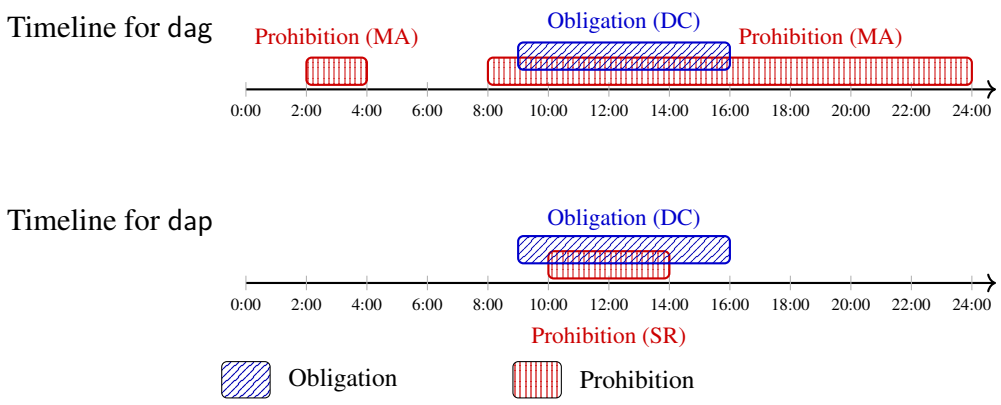


Figure 3.1: Temporal view of the use case norms, depicting when for each action obligations and prohibitions are in force: delivery at parking (dap) and delivery at gate (dag).

happen at a given time point. We thus represent the sequence of actions (the behavior of the agent) as *timed action sequences*, which are modeled as *metric-time words*. We simplify our framework by assuming that each action is instantaneous and happens at a single point in time. Actions with longer durations may be considered in future work.

With this framework in place, we now proceed to a more detailed analysis of the normative specifications of our use case. First, we can identify the following two actions to be performed by the driver:

1. Delivery at the gate, denoted by dag, cf. DC and MA;
2. Delivery at the parking, denoted by dap, cf. DC and SR.

The identified driver's actions are subject to *obligation: has to perform the action* and *prohibition: has to abstain from performing the action*:

1. Obligation: dag and dap, cf. DC;
2. Prohibition: dag, cf. MA, and dap, cf. SR.

The obligations and prohibitions are to be enacted on specific intervals of time:

1. *Between 09:00 and 16:00*: concerning the obligation on dap, cf. DC
2. *Between 10:00 and 14:00*: concerning the prohibition on dap, cf. SR
3. *Between 02:00 and 04:00 and between 08:00 and 24:00* concerning the prohibition on dag, cf. MA

Additionally, the agent's obligation on dag and dap (under DC) is in fact a choice, so she must comply with at most one of them (we will represent this in our logic as a *disjunction*).

Lastly, note that the agent is subject to all three stipulated norms (DC, SR, MA) and must comply with all of them. We will represent this in our logic as a *conjunction*.

Let us now turn our attention to our notion of conflicts. In this paper, we analyze conflicts over time. A temporal conflict arises when two or more norms apply to the same entity within overlapping time intervals but impose *contradictory or unrealistic requirements*. Unlike static conflicts, which exist regardless of time, temporal conflicts are dynamic and depend on how rules interact over time. We display in Figure 3.1 the graphical projection of the norms in the use case across a 24-hour timeline, emphasizing conflicts from an intuitive visual standpoint. To distinguish the types of normative constraints, we use a **color- and pattern-based scheme** combined with discrete hour markers:

- **(Obligations)** — **Blue hatched rectangles** represent intervals during which an action is **mandated** by a normative clause.
  - Example: the hatched blue band above the timeline denotes an obligation to deliver at the parking location (dap) between **09:00–16:00**, as imposed by DC.
- **(Prohibitions)** — **Red striped rectangles** indicate intervals during which an action is **forbidden** by a normative clause.
  - Example: the striped red band below the timeline shows that deliveries at the parking location (dap) are prohibited between **10:00–14:00** under SR.
- **Conflicting time points** occur when **blue** and **red** rectangles overlap, meaning that an action is simultaneously **obligated** and **forbidden** at the same time points.
  - Example conflict (DC vs. SR): the overlap between the blue and red bands on the timeline of dap highlights the interval **10:00–14:00**, where the delivery is both required and prohibited.

A second conflict (DC vs. MA) appears on the timeline for action dag, where the obligation from DC (09:00–16:00) coincides with the prohibitions from MA (08:00–24:00). The conflict time points thus match precisely the obligation time points for on dag under DC, i.e: 09:00, 10:00, 11:00, 12:00, 13:00, and 14:00.

In our use case, the following principles may be applied to resolve conflicts:

- **Priority of specific rules**: The defined *Maintenance Announcement* (MA) for specific re-infrastructure repairs and the *Safety Regulations* (SR) for handling dangerous goods are likely to override broader *Delivery Clause* (DC) obligations. Consequently, the driver is not allowed to use the gate but has to comply with restricted parking times from the *Safety Regulations*.

- **Harmonization of norms**: The obligation in the *Delivery Clause* to deliver from 09:00 to 16:00 may be reinterpreted not to include the hours during which the *Safety Regulations* prohibit activities. This way, the *Delivery Clause* and the *Safety Regulations* are complied with, and unreasonable outcomes are avoided.
- **Impossibility defense**: For the delivery at the gate, the prohibition from *Maintenance Announcement* implies an objective impossibility, allowing the driver to argue discharge of obligation under doctrines such as *force majeure* or *frustration of purpose*. Compliance is still feasible for delivery at parking, so no such defense is applicable.
- **Hierarchy of norms**: If the *Safety Regulations* and *Maintenance Announcement* are regulatory mandates (public law), they override the private *Delivery Clause* from the goods delivery contract under the principle *publicum ius privatorum pactis mutari non potest* ("private agreements cannot alter public law").

In practice, the driver has the option to adopt the following **practical strategy**:

1. Avoid delivery at gate entirely: The prohibition from the *Maintenance Announcement* between 08:00 to 24:00 makes gate deliveries impossible.
2. Schedule delivery at parking only during permitted times: The driver must choose to deliver to the parking area either between 09:00 and 10:00 or between 14:00 and 16:00 to comply with both the *Delivery Clause* and the *Safety Regulations*.

In the following sections, we introduce the formal definitions and notations necessary to construct a logical language for analyzing potential normative conflicts. Building on this foundation, we develop a reasoning framework grounded in the syntax and semantics of the logic. This framework serves three key purposes: to determine whether a viable strategy exists for complying with the normative specification; (ii) to formally identify conflicting time points where some norms from the normative system are incompatible; and (iii) to derive a refined version of the specification that eliminates such conflicts while preserving the normative intent, thereby providing the agent with a clear and streamlined formula to support its planning and reasoning. Throughout the paper, we illustrate these components using a running use case.

3.3 The Micro Metric-Time Normative Logic (μMTNL)

Following the use case, our intuition suggests that a conflict manifests when an action is both required and prohibited at the same point in time. We refer to such a situation as a *deontic conflict*. Additionally, a second notion of conflict, called an *ontic conflict*, arises

when two distinct actions are required to occur at the same time, which is infeasible under the assumption that at most one action can occur at any time point.

The above explanation of conflict is based on intuition rather than formal grounds. To establish a formal definition, we first introduce a normative logic, reduced to the essential operators necessary for both modeling the key aspects of our use case and formalizing our notion of conflict.

We start by defining the micro metric-time normative logic (μMTNL), which can capture clauses of normative specifications and their combinations in terms of logical formulas. We first define the syntax and semantics of the logic. The logic is designed so that a formula defines a language containing those metric-time words satisfying it. As pointed out in the introduction, we establish how conflicts and the satisfiability of (sub) formulas correspond. To this end, we also study the satisfiability problem of μMTNL.

3.3.1 Syntax of μMTNL

The *micro metric-time normative logic* focuses on *metric time*, where norms are equipped with canonical interval sets from $\mathcal{T}$ (Definition 2) as timed constraints. A formula ϕ within the logic is constructed by combining literals: A literal (or atomic norm) is formed by three components, (i) a deontic operator that could be an obligation (O) or a prohibition (F), (ii) an action from Σ, and (iii) a set of intervals $\mathfrak{I}$.

The literal $O^{\mathfrak{I}}(\mathsf{a})$ intuitively indicates that "the agent has an obligation to perform action a at least once within the interval set $\mathfrak{I}$". To avoid the well known procrastination effect, studied in [BDDM04] and [BBT08], we enforce that the specified set of intervals for obligations is not formed by an infinite interval, which enforces a clear deadline on the performance of the obliged action and avoids procrastination. The literal $F^{\mathfrak{I}}(\mathsf{a})$ indicates "the agent is prohibited from performing action a at any time point from the interval set $\mathfrak{I}$". We do not impose the same constraint of a *finite* deadline for prohibitions as a prohibition does not lead to any procrastination and additionally, some prohibitions, in practice, may be specified to start at some time point k and last *forever*, which could be specified by the interval set $\{[k,\infty]\}$. The composition of literals is obtained by the *Boolean operators*: conjunction ($\sqcap$) and disjunction ($\sqcup$).

Definition 16 (Syntax of μMTNL)**.** *Given a set of actions Σ, the syntax of μMTNL formulas over Σ is defined inductively via the following grammar:*

$$\phi \quad ::= \quad O^{\mathfrak{I}^*}(\mathsf{a}) \mid F^{\mathfrak{I}}(\mathsf{a}) \mid \phi \sqcup \phi \mid \phi \sqcap \phi$$

where $\mathfrak{I}^*$ is an interval set formed by finite intervals ranging over elements of $\mathcal{T}$, $\mathfrak{I}$ ranges over elements of $\mathcal{T}$, and a ranges over Σ. When Σ is clear from the context, we may omit it.

Example 16. *To illustrate our logic, let us transform the textual representation of norms from our use case to formulas from* μMTNL. *First, the Delivery Clause in Table Table 3.1, where an agent can choose between delivering at the gate and delivering at the parking between 9:00 and 16:00, and could be represented in* μMTNL *as:*

$$\mathsf{DC} := O^{\{[9,16]\}}(\mathsf{dag}) \sqcup O^{\{[9,16]\}}(\mathsf{dap}).$$

The Safety Regulation norm from Table Table 3.1 prohibits the delivery of dangerous goods at the parking between 10:00 and 14:00, and could be expressed in μMTNL *as follows:*

$$\mathsf{SR} := F^{\{[10,14]\}}(\mathsf{dap}).$$

Lastly, the Maintenance Announcement in Table Table 3.2, prohibits the delivery at the gate between 2:00 and 04:00 and between 8:00 and 24:00, and could be written in our logic as follows:

$$\mathsf{MA} := F^{\{[2,4]\},\{[8,24]\}}(\mathsf{dag}).$$

Thus, our use case normative system would be represented as: $\mathsf{UC} := \mathsf{DC} \sqcap \mathsf{SR} \sqcap \mathsf{MA}$, *resulting in:*

$$\mathsf{UC} = \left(O^{\{[9,16]\}}(\mathsf{dag}) \sqcup O^{\{[9,16]\}}(\mathsf{dap})\right) \sqcap F^{\{[10,14]\}}(\mathsf{dap}) \sqcap F^{\{[2,4],[8,24]\}}(\mathsf{dag}).$$

We now introduce two syntactic metrics for formulas, starting with the size of a formula, which reflects the number of symbolic elements that must be stored in memory to reason about it.

Definition 17 (Size of a μMTNL Formula)**.** *Let* ϕ *be a formula from* μMTNL. *The* size *of the formula* ϕ *written as* $|\phi|$ *is defined inductively as:*

$$|\phi| = \begin{cases} 2 + 2 \times \mathit{card}(\mathfrak{I}) & \text{if } \phi = O^{\mathfrak{I}}(\mathsf{a}) \text{ or } \phi = F^{\mathfrak{I}}(\mathsf{a}), \\ |\phi_1| + |\phi_2| + 1 & \text{if } \phi = \phi_1 \sqcap \phi_2 \text{ or } \phi = \phi_1 \sqcup \phi_2. \end{cases}$$

where $\mathsf{card}(\mathfrak{I})$ *denotes the number of intervals in the set* $\mathfrak{I}$.

The size of a literal obligation or prohibition is determined by: the obligation or prohibition symbol and the action name, which is a fixed length of two symbols; additionally, one has to keep two numeric bounds per interval from the interval set. Hence, the total size of a literal is given by $2 + 2 \times |\mathfrak{I}|$. For the case $\phi_1 \bowtie \phi_2$, where the boolean operator $\bowtie \in \{\sqcap, \sqcup\}$, the total size is given as the size of the boolean operator plus the size of both ϕ_1 and ϕ_2.

The next syntactic metric we define is called the *duty measure*. It estimates the degree of duty a formula imposes on the agent. The measure is counting the number of obligation literals forming the formula.

Definition 18 (Duty Measure for a μMTNL Formula)**.** *Let ϕ be a formula in* μMTNL. *The* duty measure *ϕ, denoted by $\eta(\phi)$, corresponds to the number of obligation literals forming the formula and is defined inductively as follows:*

$$\eta(\phi) = \begin{cases} 1 & \text{if } \phi = O^{\mathfrak{I}}(\mathsf{a}), \\ 0 & \text{if } \phi = F^{\mathfrak{I}}(\mathsf{a}), \\ \eta(\phi_1) + \eta(\phi_2) & \text{if } \phi = \phi_1 \sqcap \phi_2 \text{ or } \phi = \phi_1 \sqcup \phi_2. \end{cases}$$

Example 17 (Size and Duty Measure for Example 16)**.** *We have* $\mathsf{UC} := \mathsf{DC} \sqcap \mathsf{SR} \sqcap \mathsf{MA}$. *Applying the definitions for the size and duty measure we obtain:*

- *For* $\mathsf{DC} := O^{\{[9,16]\}}(\mathsf{dap}) \sqcup O^{\{[9,16]\}}(\mathsf{dap})$, *we have* $|\mathsf{DC}| = (2+2)+1+(2+2) = 9$ *and* $\eta(\mathsf{DC}) = 2$.
- *For* $\mathsf{SR} := F^{\{[10,14]\}}(\mathsf{dap})$, *we have:* $|\mathsf{SR}| = 4$ *and* $\eta(\mathsf{SR}) := 0$.
- *For* $\mathsf{MA} := F^{\{[2,4],[8,24]\}}(\mathsf{dag})$ *we have* $|\mathsf{MA}| = 6$ *and* $\eta(\mathsf{MA}) = 0$.

Thus, $|\mathsf{UC}| = |\mathsf{DC}| + 1 + |\mathsf{SR}| + 1 + |\mathsf{MA}| = 9+1+4+1+6 = 21$ *and,* $\eta(\mathsf{UC}) = \eta(\mathsf{DC}) + \eta(\mathsf{SR}) + \eta(\mathsf{MA}) = 2+0+0 = 2$.

μMTNL is labeled as micro because it is a simple logic that focuses solely on the deontic operators *obligations* and *prohibitions* and their combination using conjunction and disjunction. The logic lacks temporal operators, conditionals, and permission deontic operators, but it is enough to highlight our ideas on the notion of conflict.

3.3.2 Semantics of μMTNL

We adopt a trace-based semantics over finite timed words (Definition 7). This choice aligns directly with punctual, action-centric constraints: formulas are evaluated on sequences of timestamped actions, not on state valuations.

Why timed words rather than Kripke frames? We use timed words rather than Kripke frames to *impose atomicity constraints*: (i) each event is instantaneous, (ii) timestamps are strictly increasing, and (iii) at most one action occurs at any time point act as *built-in frame conditions* for our conflict definitions. This is in the same spirit in which Kripke justifies modal axioms (T, S4, S5) by imposing properties on the accessibility relation. Timed words can also be viewed as a highly restricted Kripke semantics: worlds are event indices $0,\ldots,n$, the relation R links each index to its immediate successor, and the labeling function is $L(i) = (a_i, t_i)$. Linearity, irreflexivity, and functional succession then hold by construction, which is why we work directly with timed words in what follows.

Timed word semantics μMTNL We adopt a trace-based interpretation over finite timed words. A formula constrains which action–time pairs may occur in a trace and how often. The satisfaction relation below is defined by structural induction.

Timed words semantics

Definition 19 (Satisfaction relation)**.** *Given a finite metric-time word* $\tau \in \mathbb{T}$ *and a formula* ϕ *from* μMTNL*, the satisfaction relation* $\tau \vDash \phi$ *(read as"the metric-time word* τ *satisfies the formula* ϕ*") is defined inductively on the structure of the formula* ϕ *as follows:*

$$
\begin{aligned}
&\tau \vDash O^{\mathfrak{I}}(\mathsf{a}) && \textit{iff} && \exists t \trianglelefteq \mathfrak{I} : \rho(\tau, t) = \mathsf{a}.\\
&\tau \vDash F^{\mathfrak{I}}(\mathsf{a}) && \textit{iff} && \forall t \trianglelefteq \mathfrak{I} : \rho(\tau, t) \neq \mathsf{a}.\\
&\tau \vDash \phi \sqcup \phi' && \textit{iff} && \tau \vDash \phi \textit{ or } \tau \vDash \phi'.\\
&\tau \vDash \phi \sqcap \phi' && \textit{iff} && \tau \vDash \phi \textit{ and } \tau \vDash \phi'.
\end{aligned}
$$

The semantics defines how to satisfy an obligation literal using the action lookup function ρ (cf. Definition 10), which requires that the obliged action has been performed at least once with the specified timed conditions. Prohibition forbids any occurrence of the given action within the specified interval set. Conjunction ($\sqcap$) adds more duties, and disjunction ($\sqcup$) offers the agent the choice of one duty instead of another.

Example 18. *Consider the metric-time word* $\tau := \langle(\mathsf{dag}, 1)\ (\mathsf{pick\_up}, 3)\ (\mathsf{dap}, 11)\rangle$*. We show below that this metric-time word does not satisfy the* $\mathsf{DC} \sqcap \mathsf{SR} \sqcap \mathsf{MA}$ *as specified in Example 16:*

- $\langle(\mathsf{dag}, 1)\ (\mathsf{pick\_up}, 3)\ (\mathsf{dap}, 11)\rangle \vDash O^{\{[9,16]\}}(\mathsf{dap})$ *as the event* $(\mathsf{dap}, 11)$ *corresponds to timed condition of the obligation* $11 \trianglelefteq \{[9, 16]\}$.
- $\langle(\mathsf{dag}, 1)\ (\mathsf{pick\_up}, 3)\ (\mathsf{dap}, 11)\rangle \nvDash F^{\{[10,14]\}}(\mathsf{dap})$ *as the event* $(\mathsf{dap}, 11)$ *is a violation of the timed condition of the prohibition as* $11 \trianglelefteq \{[10, 14]\}$.
- $\langle(\mathsf{dag}, 1)\ (\mathsf{pick\_up}, 3)\ (\mathsf{dap}, 11)\rangle \vDash O^{[9,16]}(\mathsf{dap}) \sqcup O^{\{[9,16]\}}(\mathsf{dag})$.
- $\langle(\mathsf{dag}, 1)\ (\mathsf{pick\_up}, 3)\ (\mathsf{dap}, 11)\rangle \nvDash F^{\{[10,14]\}}(\mathsf{dap}) \sqcap F^{\{[2,4],[8,24]\}}(\mathsf{dag})$.
- $\langle(\mathsf{dag}, 1)\ (\mathsf{pick\_up}, 3)\ (\mathsf{dap}, 11)\rangle \nvDash \mathsf{DC} \sqcap \mathsf{SR} \sqcap \mathsf{MA}$.

Remark 2 (Kinds of obligations and probihitions)**.** *Note that there are many different ways to capture the idea of obligation and prohibition in ways such as achievement and maintenance related to the timed interpretation as defined in [BDDM04, GR11]: Achievement refers to successfully achieving a task at least once before a specific deadline, like submitting a report on time. Once this task is accomplished, the achievement constraint*

is relieved. In contrast, maintenance involves consistently upholding a property over a specific period *of time. It is not enough to do it just once; it must be sustained over time. Our definition of obligation fits the definition of* achievement obligations *from [DK98]. Likewise, we understand our definition of prohibition as* negative achievement *over a set of time points, despite its similarity with maintenance prohibition, as both require full coverage on the prohibited specified timed points. The key difference resides in the fact that the prohibited event must* never *occur rather than maintaining a continuous state.*

Remark 3 (On the absence of negation)**.** μMTNL *is a positive logic. We do not define negation for this logic. A* straightforward[2] *way to define the semantics of negation, where the syntax of a negated formula ϕ is written as $\neg\phi$ and would be defined as follows:*

$$\tau \vDash \neg\phi \text{ iff } \tau \nvDash \phi.$$

By substituting ϕ by an obligation say $O^{\mathfrak{I}}(\mathsf{a})$, we obtain:

$$\tau \vDash \neg O^{\mathfrak{I}}(\mathsf{a}) \text{ iff } \tau \nvDash O^{\mathfrak{I}}(\mathsf{a}).$$

When substituted by the semantics of an obligation, this reduces to :

$$\tau \vDash \neg O^{\mathfrak{I}}(\mathsf{a}) \text{ iff } \tau \nvDash \exists t \subseteq \mathfrak{I} : \rho(\tau,t) = \mathsf{a}.$$

We know that by negating the first order formula, the definition becomes:

$$\tau \vDash \neg O^{\mathfrak{I}}(\mathsf{a}) \text{ iff } \tau \vDash \forall t \subseteq \mathfrak{I} : \rho(\tau,t) \neq \mathsf{a}, \text{ iff } \tau \vDash F^{\mathfrak{I}}(\mathsf{a}).$$

We obtain that the semantics of $\neg O^{\mathfrak{I}}(\mathsf{a})$ matches the semantics of $F^{\mathfrak{I}}(\mathsf{a})$. This is counter-intuitive to our understanding of obligation and prohibition: not being obliged to perform an action a *does not necessarily mean that performing* a *is forbidden.*

Another way of introducing negation could be at the level of actions. Although it could be tried to read the notation $\neg a$ as a syntactic construction denoting the "negation of action a*", doing so imposes a semantic flaw: actions differ from propositions by their very nature; as they denote occurrences, events, or behaviors; and therefore negating them means more than taking the logical complement. We refrain from specifying negation on the action as in other logics and other formal languages where actions are first-class citizens such as [Hoa78, HKT01, Hor01, DGV+13]. Therefore, we represent negative constraints by using explicit prohibition modalities (e.g., $F^{\mathfrak{I}}(\mathsf{a})$), which precisely express the intended normative requirements and temporal constraints. This distinguishes between ontological absence (an action not occurring) and deontic force (an action being forbidden), maintaining a clear distinction between event structure and norm interpretation.*

[2]This definition is used in other logics such as Linear Temporal Logic, Dyadic Deontic Logic,...

Extensional semantics

Another perspective of word semantics is known as extensional semantics, in which a formula is interpreted to be a set of acceptable behaviors. In such semantics, also known as model enumeration, the semantics of a formula are given in terms of the finite timed words that are models of its specifications. Specifically, the semantics of a formula are given by the set of words that adhere to its normative requirements. Formally, the language of a formula ϕ, denoted by $[\![\phi]\!]$, is the set of all metric-time words that satisfy ϕ. In planning or normative settings, this set contains all compliant execution plans, which concretely delineate the possible successful agent behaviors under the given normative specification.

Definition 20 (Language of a Formula)**.** *The* language *of a formula ϕ from μMTNL written by $[\![\phi]\!]$ is a set (which may be infinite) of finite metric-time words that satisfy the formula.*

$$[\![\phi]\!] := \{\tau \mid \tau \vDash \phi\}.$$

Lemma 3 (Characterization of the Language of a Formula)**.** *Let ϕ be a formula in μMTNL. The language of the formula, denoted $[\![\phi]\!]$, can be characterized as follows:*

$$[\![O^{\mathfrak{I}}(\mathsf{a})]\!] = \{\tau \mid \exists t : t \in \mathfrak{I} \text{ and } \rho(\tau,t) = \mathsf{a}\}.$$
$$[\![F^{\mathfrak{I}}(\mathsf{a})]\!] = \{\tau \mid \forall t \in \mathfrak{I} : \rho(\tau,t) \neq \mathsf{a}\}.$$
$$[\![\phi \sqcup \phi']\!] = [\![\phi]\!] \cup [\![\phi']\!].$$
$$[\![\phi \sqcap \phi']\!] = [\![\phi]\!] \cap [\![\phi']\!].$$

The language of a literal obligation is the set of metric-time words where the obligated action occurs at least once within the specified time interval. In contrast, the language of a literal prohibition includes the set of metric-time words where the prohibited action does not take place within the given time constraints specified in the interval set. The language of a disjunction $\phi \sqcup \phi'$ is the union of the languages of the individual formulas, and the language of a conjunction $\phi \sqcap \phi'$ is the intersection of their respective languages.

3.3.3 Semantic Properties of μMTNL

In this subsection, we explain and analyze some specific semantic properties of μMTNL. Most of our discussion is concentrated on the influence of logical operators on formula satisfiability in the sense of how unsatisfiable "parts" of formulae influence compound expressions. We also examine how unsatisfiable components of a formula can be eliminated without altering its overall meaning. To achieve this, we define a representation of literals as combinations of more elementary ones.

We begin by stating the fundamental algebraic equivalence properties of the logical operators, namely: commutativity, associativity, and distributivity, presented in Lemma 4. Equivalence between two formulas ϕ and ϕ' is written by $\phi \equiv \phi'$, when their duty classes are equal, i.e $[\![\phi]\!] = [\![\phi']\!]$. They are identical to those in classical logic and other standard logical systems. These properties play a crucial role in reducing representations to normal forms, which in effect facilitates the detection of unsatisfiable sub-formulas and enables systematic reasoning regarding the inconsistencies of a specification.

Lemma 4 (Algebraic Equivalences of μMTNL Operators)**.** *Let ϕ_1, ϕ_2, ϕ_3 be formulas in μMTNL. Then the following equivalences hold:*

$$\phi_1 \sqcup \phi_2 \equiv \phi_2 \sqcup \phi_1 \qquad \textit{(Comm1)}$$
$$\phi_1 \sqcap \phi_2 \equiv \phi_2 \sqcap \phi_1 \qquad \textit{(Comm2)}$$
$$\phi_1 \sqcup (\phi_2 \sqcup \phi_3) \equiv (\phi_1 \sqcup \phi_2) \sqcup \phi_3 \qquad \textit{(Assoc1)}$$
$$\phi_1 \sqcap (\phi_2 \sqcap \phi_3) \equiv (\phi_1 \sqcap \phi_2) \sqcap \phi_3 \qquad \textit{(Assoc2)}$$
$$\phi_1 \sqcap (\phi_2 \sqcup \phi_3) \equiv (\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3) \qquad \textit{(Distr1)}$$
$$\phi_1 \sqcup (\phi_2 \sqcap \phi_3) \equiv (\phi_1 \sqcup \phi_2) \sqcap (\phi_1 \sqcup \phi_3) \qquad \textit{(Distr2)}$$

Proof. The proof follows from the Lemma 3, recall that $\phi \equiv \phi'$ if and only if $[\![\phi]\!] = [\![\phi']\!]$. We proceed case by case:

- **(Comm1)** and **(Comm2)**: Follows from the fact that set union and intersection are commutative operations:

$$[\![\phi_1 \sqcup \phi_2]\!] = [\![\phi_1]\!] \cup [\![\phi_2]\!] = [\![\phi_2]\!] \cup [\![\phi_1]\!] = [\![\phi_2 \sqcup \phi_1]\!].$$

- **(Assoc1)** and **(Assoc2)**: These follow from associativity of union and intersection over sets:

$$[\![\phi_1 \sqcup (\phi_2 \sqcup \phi_3)]\!] = [\![\phi_1]\!] \cup ([\![\phi_2]\!] \cup [\![\phi_3]\!]) = ([\![\phi_1]\!] \cup [\![\phi_2]\!]) \cup [\![\phi_3]\!] = [\![(\phi_1 \sqcup \phi_2) \sqcup \phi_3]\!].$$

- **(Distr1)**: Distributivity of intersection over union:

$$\begin{aligned}[\![\phi_1 \sqcap (\phi_2 \sqcup \phi_3)]\!] &= [\![\phi_1]\!] \cap ([\![\phi_2]\!] \cup [\![\phi_3]\!]) \\ &= ([\![\phi_1]\!] \cap [\![\phi_2]\!]) \cup ([\![\phi_1]\!] \cap [\![\phi_3]\!]) \\ &= [\![(\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3)]\!].\end{aligned}$$

- **(Distr2)**: Distributivity of union over intersection:

$$\begin{aligned}[\![\phi_1 \sqcup (\phi_2 \sqcap \phi_3)]\!] &= [\![\phi_1]\!] \cup ([\![\phi_2]\!] \cap [\![\phi_3]\!]) \\ &= ([\![\phi_1]\!] \cup [\![\phi_2]\!]) \cap ([\![\phi_1]\!] \cup [\![\phi_3]\!]) \\ &= [\![(\phi_1 \sqcup \phi_2) \sqcap (\phi_1 \sqcup \phi_3)]\!].\end{aligned}$$

Hence, in all cases and by operations on duty language, the left-hand side and the right-hand side formulas denote the same set of acceptable timed words. Thus, the formulas are semantically equivalent. □

We study now some normal forms. The term *normal form*, in the context of formulas of logic, is commonly used to indicate that a formula has certain syntactic properties. We recall the two known normal forms on the alternation between conjunctions and disjunctions.

Definition 21 (Conjunction Normal Form (*CNF*))**.** *A* clause *is a disjunction of literals. A formula ϕ is in* conjunctive normal form*, short CNF, if it is a conjunction of clauses:*

$$\phi \text{ is CNF if and only if } \phi := \bigsqcap_i \Big(\bigsqcup_j \ell_{i,j} \Big).$$

Note that a formula in *CNF* can contain zero conjunctions and/or zero disjunctions per clause, but the order of alternation must be a disjunction of conjunctions of literals.

Definition 22 (Disjunctive Normal Form (*DNF*))**.** *A* product term *is a conjunction of literals. A formula ϕ is in* disjunctive normal form*, short DNF, if it is a disjunction of product terms:*

$$\phi \text{ is DNF if and only if } \phi := \bigsqcup_i \Big(\bigsqcap_j \ell_{i,j} \Big).$$

Note that a formula in *DNF* can contain zero conjunctions and/or zero disjunctions per product term, but the order of alternation must be a disjunction of conjunctions of literals.

Example 19. *Let us recall parts of our use case and check if they are in CNF and/or DNF.*

1. *As* $\mathsf{SR} = F^{\{[10,14]\}}(\mathsf{dap})$*, and according to our definition* SR *is both in CNF and DNF as the formula contains zero conjunctions and zero disjunctions. The same applies for* $\mathsf{MA} = F^{\{[2,4]\},\{[8,24]\}}(\mathsf{dag})$.
2. *As* $\mathsf{DC} = O^{\{[9,16]\}}(\mathsf{dag}) \sqcup O^{\{[9,16]\}}(\mathsf{dap})$ *and according to our definitions,* DC *is in DNF as two terms formed by one literal and is also is in CNF formed by one clause of two literals.*

3. *For the overall use case, we have* $\mathsf{UC} = \mathsf{DC} \sqcap \mathsf{SR} \sqcap \mathsf{MA}.\mathsf{UC}$ *is in CNF as a 3-clause conjunction where the first is formed by two literal obligations and the two remaining clauses are formed by single literals. The formula is not in DNF as the order of alternation of conjunction and disjunction does not match the pattern description of the format.*

We saw that the overall use case is not in *DNF*. We proceed now to show that any formula from μMTNL could be normalized to *DNF*. Additionally, we prove some properties of the formula resulting from normalization.

Lemma 5 (Normalization to *DNF*)**.** *For every formula* $\phi \in \mu\mathsf{MTNL}$*, there exists a* $\phi' \in \mu\mathsf{MTNL}$ *such that:*

(i) $\phi' \equiv \phi$*,*

(ii) ϕ' *is in DNF,*

(iii) $\eta(\phi) \leq \eta(\phi')$*, and*

(iv) for $\phi' := \bigsqcup \phi'_i$*, each* ϕ'_i *satisfies* $\eta(\phi'_i) \leq \eta(\phi)$*.*

Proof. The construction proceeds by systematically applying the distributivity equivalence from Lemma 4, specifically:

$$\phi_1 \sqcap (\phi_2 \sqcup \phi_3) \equiv (\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3) \quad \textbf{(Distr1)},$$

which we orient into the rewriting rule:

$$\phi_1 \sqcap (\phi_2 \sqcup \phi_3) \to (\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3) \quad \textbf{(Distr1L)}.$$

Starting from ϕ, we repeatedly apply this rule to push disjunctions outward and distribute conjunctions inward. This process terminates with a formula ϕ' that is disjunctive normal form (*DNF*), thus establishing (ii). Equivalence is preserved at each rewriting step, which establishes (i).

For (iii), consider the effect of applying the distributivity rule:

$$\phi_1 \sqcap (\phi_2 \sqcup \phi_3) \to (\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3).$$

Let us compare the number of obligations measured by the η function. We have, by the definition of the function:

$$\eta((\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3)) = 2 \times \eta(\phi_1) + \eta(\phi_2) + \eta(\phi_3),$$

and

$$\eta(\phi_1 \sqcap (\phi_2 \sqcup \phi_3)) = \eta(\phi_1) + \eta(\phi_2) + \eta(\phi_3).$$

This shows that distributing a conjunction over a disjunction results in a duplication of ϕ_1, effectively doubling the number of obligation literals it contributes. As such, each application of the distributivity rule can increase the total number of obligation literals in the formula.

When applied recursively, over nested conjunctions or disjunctions, this process propagates duplication and can cause an exponential growth in the number of obligation literals. Therefore, this justifies the inequality: $\eta(\phi) \leq \eta(\phi')$, which captures the potential syntactic blow-up caused by structural rewriting.

For (iv), we refine the reasoning to show that each disjunct in the resulting formula contains no more obligation literals than the original formula.

Suppose the normalized formula $\phi' = \bigsqcup \phi'_i$ is obtained from ϕ by repeated application of the distributivity rule:

$$\phi_1 \sqcap (\phi_2 \sqcup \phi_3) \rightarrow (\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3).$$

Each ϕ'_i is thus a conjunction of subformulas that results from one of the paths taken during the distributive unfolding. For instance, from the rule above, we know:

$$\eta((\phi_1 \sqcap \phi_2)) = \eta(\phi_1) + \eta(\phi_2),$$

and similarly for $\phi_1 \sqcap \phi_3$. Therefore, in the resulting disjunction:

$$\phi' = (\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3),$$

we have two disjuncts, each of which contains at most $\eta(\phi_1) + \eta(\phi_2)$ or $\eta(\phi_1) + \eta(\phi_3)$ obligation literals, respectively. Which are both less or equal than $\eta(\phi_1) + \eta(\phi_2) + \eta(\phi_3)$ from the original formula.

Critically, although the total size $\eta(\phi')$ may increase (due to duplication), each individual disjunct ϕ'_i comes from a single branch of the distributive rewriting and does not aggregate all duplicated parts. Thus, for every i:

$$\eta(\phi'_i) \leq \eta(\phi),$$

Because the rewriting step does not introduce new literals, it only pushes existing ones. This ensures that no disjunct grows beyond the syntactic size of the original formula, even if the whole disjunction becomes longer after a certain number of iterations. This observation supports point (iv).

□

Example 20 (*DNF* normalization of UC)**.** *We continue with Example 19, and demonstrate the normalization of* UC*. Using the first distributivity law once, we obtain:* $\mathsf{UC}' \equiv \mathsf{UC}$*, where* UC' *is in DNF. Consider:* $\mathsf{UC}'_1 := O^{\{[9,16]\}}(\mathsf{dag}) \sqcap \mathsf{SR} \sqcap \mathsf{MA}, \mathsf{UC}'_2 := O^{\{[9,16]\}}(\mathsf{dap}) \sqcap \mathsf{SR} \sqcap \mathsf{MA}$*. Then:* $\mathsf{UC}' := \mathsf{UC}'_1 \sqcup \mathsf{UC}'_2$*.* UC' *is in DNF and equivalent to* $\mathsf{DC} \sqcap \mathsf{SR} \sqcap \mathsf{MA}$*, with:* $\eta(\mathsf{UC}) \leq \eta(\mathsf{UC}')$ *and* $\eta(\mathsf{UC}'_i) \leq \eta(\mathsf{UC})$ *for each* $i = 1, 2$*.*

We now turn to a second normal form aimed at refining the temporal granularity of formulas. Lemma 6 introduces two equivalences for *literal compression*: the first, **ObDis**, shows that obligation literals referring to the same action are disjunction-composable; the second, **FConj**, captures the analogous property for prohibition literals under conjunction.

Lemma 6 (Literal Compression Equivalences)**.** *Let* $\mathfrak{I}$ *and* $\mathfrak{I}'$ *be interval sets, and let* a *be an action. Then the following equivalences hold in* μMTNL*:*

$$O^{\mathfrak{I}}(\mathsf{a}) \sqcup O^{\mathfrak{I}'}(\mathsf{a}) \equiv O^{\mathfrak{I} \cup \mathfrak{I}'}(\mathsf{a}) \qquad \textit{(ObDis)}$$

$$F^{\mathfrak{I}}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a}) \equiv F^{\mathfrak{I} \cup \mathfrak{I}'}(\mathsf{a}) \qquad \textit{(FConj)}$$

Proof. Both equivalences follow from the extensional semantics of μMTNL, where disjunction corresponds to set union, conjunction to set intersection, and the obligation and prohibition literals are interpreted respectively via existential and universal quantification over time points in their interval sets.

(ObDis)

$$\begin{aligned}
[\![O^{\mathfrak{I}}(\mathsf{a}) \sqcup O^{\mathfrak{I}'}(\mathsf{a})]\!] &= [\![O^{\mathfrak{I}}(\mathsf{a})]\!] \cup [\![O^{\mathfrak{I}'}(\mathsf{a})]\!] \\
&= \{\tau \mid \exists t \in \mathfrak{I} : \rho(\tau, t) = a\} \cup \{\tau \mid \exists t \in \mathfrak{I}' : \rho(\tau, t) = a\} \\
&= \{\tau \mid \exists t \in (\mathfrak{I} \cup \mathfrak{I}') : \rho(\tau, t) = a\} \quad \text{(by distributivity of } \exists) \\
&= [\![O^{\mathfrak{I} \cup \mathfrak{I}'}(\mathsf{a})]\!]
\end{aligned}$$

(FConj)

$$\begin{aligned}
[\![F^{\mathfrak{I}}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a})]\!] &= [\![F^{\mathfrak{I}}(\mathsf{a})]\!] \cap [\![F^{\mathfrak{I}'}(\mathsf{a})]\!] \\
&= \{\tau \mid \forall t \in \mathfrak{I} : \rho(\tau, t) \neq a\} \cap \{\tau \mid \forall t \in \mathfrak{I}' : \rho(\tau, t) \neq a\} \\
&= \{\tau \mid \forall t \in (\mathfrak{I} \cup \mathfrak{I}') : \rho(\tau, t) \neq a\} \quad \text{(by merging universal quantifiers)} \\
&= [\![F^{\mathfrak{I} \cup \mathfrak{I}'}(\mathsf{a})]\!]
\end{aligned}$$

□

Example 21. *These rules can be illustrated on concrete literals drawn from our use case:*

- ***ObDis***: $O^{\{[9,10]\}}(\mathsf{dag}) \sqcup O^{\{[11,16]\}}(\mathsf{dag}) \equiv O^{\{[9,16]\}}(\mathsf{dag})$,
- ***FConj***: $F^{\{[2,4]\}}(\mathsf{dag}) \sqcap F^{\{[8,24]\}}(\mathsf{dag}) \equiv F^{\{[2,4],[8,24]\}}(\mathsf{dag})$.

Conjunctions of obligations over overlapping interval sets often mask implicit structural distinctions in how normative requirements are distributed over time. In particular, when two obligations refer to the same action but cover different, partially overlapping time scopes, it becomes useful to isolate the common part from the non-overlapping remainders. The following lemma formalizes the idea that two obligations on the same action over overlapping interval sets can be achieved by performing the action once within the overlapping time points.

Lemma 7 (Obligation Conjunction Decomposition)**.** *Let $\mathfrak{I}$ and $\mathfrak{I}'$ be two overlapping and not equal interval sets. Then the conjunction of two obligations over the same action* a *can be decomposed into the disjunction of:*

1. *an obligation over the overlapping interval set $\mathfrak{I} \cap \mathfrak{I}'$, and*
2. *the conjunction of the remaining obligations over the non-overlapping parts of $\mathfrak{I}$ and $\mathfrak{I}'$.*

Formally:

$$\frac{\mathfrak{I} \cap \mathfrak{I}' \neq \emptyset \quad \textit{and} \quad \mathfrak{I}' \neq \mathfrak{I}}{O^{\mathfrak{I}}(\mathsf{a}) \sqcap O^{\mathfrak{I}'}(\mathsf{a}) \equiv O^{\mathfrak{I} \cap \mathfrak{I}'}(\mathsf{a}) \sqcup \left(O^{\mathfrak{I} \ominus (\mathfrak{I} \cap \mathfrak{I}')}(\mathsf{a}) \sqcap O^{\mathfrak{I}' \ominus (\mathfrak{I} \cap \mathfrak{I}')}(\mathsf{a}) \right)} (\textsc{ObConj})$$

Proof sketch. The obligation $O^{\mathfrak{I}}(\mathsf{a})$ requires action a over all intervals in $\mathfrak{I}$. Given that both $O^{\mathfrak{I}}(\mathsf{a})$ and $O^{\mathfrak{I}'}(\mathsf{a})$ are asserted, the effect is to require a over the union $\mathfrak{I} \cup \mathfrak{I}'$. This can be decomposed into:

- an obligation over the shared intervals $\mathfrak{I} \cap \mathfrak{I}'$, and
- a conjunction of obligations over the remaining, non-overlapping parts of each interval set: $\mathfrak{I} \ominus (\mathfrak{I} \cap \mathfrak{I}')$ and $\mathfrak{I}' \ominus (\mathfrak{I} \cap \mathfrak{I}')$.

□

Example 22 (Obligation Conjunction Decomposition)**.** *Let: $O^{\{[1,3],[5,7]\}}(\mathsf{a}) \sqcap O^{\{[2,4],[6,8]\}}(\mathsf{a})$. By Lemma 7, the formula is equivalent to:*

$$O^{\{[2,3],[6,7]\}}(\mathsf{a}) \sqcup \left(O^{\{[1,1],[5,5]\}}(\mathsf{a}) \sqcap O^{\{[4,4],[8,8]\}}(\mathsf{a}) \right).$$

Single event trace model:

The single event trace $\tau := \langle (a,2) \rangle$ satisfies both the original conjunction and the decomposition, as it performs a *at $t \in \mathfrak{I} \cap \mathfrak{I}'$.*

Lemma 8 (Prohibition Disjunction Decomposition)**.** *Let $\mathfrak{I}, \mathfrak{I}'$ be interval sets such that $\mathfrak{I} \subseteq \mathfrak{I}'$. Then the disjunction of two prohibitions over the same action* a *can be rewritten as:*

$$\frac{\mathfrak{I} \subseteq \mathfrak{I}'}{F^{\mathfrak{I}}(\mathsf{a}) \sqcup F^{\mathfrak{I}'}(\mathsf{a}) \equiv F^{\mathfrak{I}}(\mathsf{a})} \quad (\textsc{ProhibDisj})$$

Proof. We reason using the semantics of prohibition in μMTNL. A prohibition $F^{\mathfrak{I}}(\mathsf{a})$ means that the action a is forbidden at all time points in $\mathfrak{I}$.

- The disjunction $F^{\mathfrak{I}}(\mathsf{a}) \sqcup F^{\mathfrak{I}'}(\mathsf{a})$ holds whenever at least one of the two prohibitions holds.
- Since $\mathfrak{I} \subseteq \mathfrak{I}'$, if $F^{\mathfrak{I}'}(\mathsf{a})$ holds, then so does $F^{\mathfrak{I}}(\mathsf{a})$.
- Thus, in every model where the disjunction holds, $F^{\mathfrak{I}}(\mathsf{a})$ also holds.
- Conversely, if $F^{\mathfrak{I}}(\mathsf{a})$ holds, then the disjunction holds trivially.

Therefore, both formulas are semantically equivalent:

$$F^{\mathfrak{I}}(\mathsf{a}) \sqcup F^{\mathfrak{I}'}(\mathsf{a}) \equiv F^{\mathfrak{I}}(\mathsf{a}).$$

□

Example 23. *Let:* $\mathfrak{I} = \{[2,4]\}, \quad \mathfrak{I}' = \{[2,4],[5,6]\}$.

Since $\mathfrak{I} \subseteq \mathfrak{I}'$, the prohibition disjunction simplifies as:

$$F^{\{[2,4]\}}(\mathsf{a}) \sqcup F^{\{[2,4],[5,6]\}}(\mathsf{a}) \equiv F^{\{[2,4]\}}(\mathsf{a}).$$

This reflects the fact that the disjunction does not weaken the prohibition, since the smaller set $\mathfrak{I}$ is already sufficient to capture the intended restriction on a.

When specifying time-dependent norms, the choice of how to represent temporal constraints can significantly impact the syntactic size and readability of a formula. While punctual representations (i.e., using single time points) offer the finest granularity, they often lead to verbose encodings, especially when applied to longer durations or non-unifiable intervals. Conversely, more structured representations, such as intervals or sets of intervals, can convey the same semantics far more succinctly. This syntactic economy becomes especially valuable when reasoning over large normative systems or when transforming formulas into normal forms. The following remark highlights this advantage of conciseness.

Remark 4 (Conciseness of Interval Sets in Literal Representation)**.** *The way literals represent time constraints has a significant impact on the overall size and readability of formulas. Thanks to the compression equivalences (**ObDis**) and (**FConj**) in Lemma 6, multiple intervals involving the same action can be merged into a single literal whose interval set is a union of the original intervals. This reduces the syntactic overhead and improves overall readability, which facilitates the subjects of the normative system to get a more straightforward interpretation.*

Consider, for example, the literal $F^{[2,4],[8,24]}(\mathsf{dag})$*, which expresses a prohibition over two time ranges. As shown in Table Table 3.3, this specification can be written in multiple ways—with varying levels of granularity. We compare three syntactic alternatives: using single time points, using individual intervals, and using a set of intervals. The table highlights that the interval set representation is by far the most concise, requiring fewer symbols to express the same semantics.*

This observation reinforces the value of using interval sets in their compact form, without decomposing them into punctual intervals unless required (e.g., during bounded normalization). This strategy preserves expressiveness while minimizing the computational and notational footprint.

Representation	**Formula (expressing** MA**)**	**Size**
Single time points	$\bigsqcap_{i\in\{2,3,4,8,\ldots,24\}} F^{\{[i,i]\}}(\mathsf{dag})$	99
Intervals	$F^{\{[2,4]\}}(\mathsf{dag}) \sqcap F^{\{[8,24]\}}(\mathsf{dag})$	9
Interval sets	$F^{\{[2,4],[8,24]\}}(\mathsf{dag})$	6

Table 3.3: Conciseness comparison for $\mathsf{MA} := F^{\{[2,4]\},\{[8,24]\}}(\mathsf{dag})$

These considerations conclude our overview of the semantic behavior of logical operators and the structural properties of literals. The equivalences we have introduced provide essential tools for formula normalization and simplification, laying the foundation for constructive reasoning about satisfiability and conflict.

3.3.4 The Satisfiability Problem in μMTNL

The satisfiability problem plays a central role in assessing whether a timed normative specification is operationally meaningful. Before analyzing conflicts, we must first determine whether a formula admits at least one compliant execution trace. In the context of μMTNL, satisfiability captures the fundamental feasibility of fulfilling all obligations while respecting all prohibitions under metric time constraints.

This subsection formalizes the satisfiability problem for our logic. We establish a tight bound on the size of witnessing timed words, position the problem within classical complexity theory, demonstrate the resolution procedure as well as its implementation, and introduce rules for composition with unsatisfiable formulas. These results provide the computational foundation for the subsequent conflict analysis, ensuring that the relationship between satisfiability and conflict is well-defined.

Definition and complexity

We define the unsatisfiability problem in a classical way for model basic logics where unsatisfiability relates to the inexistance of a trace satisfying a formula.

Definition 23 (Satisfiability)**.** *A formula ϕ is* satisfiable*,* sat *for short, if and only if there exists a metric-time word τ that satisfies ϕ:*

$$\phi \textit{ is } \mathsf{sat} \textit{ iff } \exists \tau : \tau \vDash \phi.$$

A formula is unsatisfiable *written* unsat *if and only if the formula is not satisfiable.*

Another possible condition for satisfiability is that the language of the formula is non-empty.

Remark 5 (Satisfiability as language definition)**.** *A formula ϕ is satisfiable if and only if the language of ϕ is not empty:*

$$\phi \textit{ is } \mathsf{sat} \textit{ iff } [\![\phi]\!] \neq \emptyset.$$

Now we study the relation between the size of a *minimal satisfying trace* in terms of the structure of the formula, more precisely to its duty measure as defined in Definition 18.

Lemma 9 (Satisfiability Size Bound)**.** *For any formula ϕ from μ*MTNL*, if ϕ is satisfiable then there exists a metric-time word τ with at most $\eta(\phi)$ events that satisfies ϕ:*

$$\textit{if } \phi \textit{ is } \mathsf{sat} \textit{ then } \exists \tau : |\tau| \leq \eta(\phi) \wedge \tau \vDash \phi.$$

This lemma ensures that a finite search space is sufficient for checking satisfiability, which impacts the complexity class we study later on.

Proof. By Lemma 5, we know that any formula ϕ can be rewritten as a disjunction of conjunctions of literals:

$$\phi \equiv \bigsqcup_{i=1}^{n} \phi_i.$$

with every disjunction clause $\phi_i := \bigsqcap_{j=1}^{m_i} \ell_j$ where each literal ℓ_j is either an obligation $O^{\mathfrak{I}}(\mathsf{a})$ or a prohibition $F^{\mathfrak{I}}(\mathsf{a})$. If ϕ is satisfiable, then at least one disjunction clause (i.e., one conjunction of literals) is satisfiable.

Each clause ϕ_i can be rearranged into a conjunction of obligation literals and a conjunction of prohibition literals. We refer to $\mathcal{F}$ as the conjunction of prohibition literals.

$$\phi_i \equiv O^{\mathfrak{I}_1}(\mathsf{a_1}) \sqcap \cdots \sqcap O^{\mathfrak{I}_n}(\mathsf{a_n}) \sqcap \underbrace{F^{\mathfrak{I}'_1}(\mathsf{b_1}) \sqcap \cdots \sqcap F^{\mathfrak{I}'_m}(\mathsf{b_m})}_{\mathcal{F}}.$$

By definition, we know that for any obligation $O^{\mathfrak{I}}(\mathsf{a})$, a single event trace (a,t) with $t \subseteq \mathfrak{I}$ is enough to satisfy the formula:

$$\forall a, \mathfrak{I}: \quad \text{if } t \subseteq \mathfrak{I} \text{ then } \langle (a,t) \rangle \vDash O^{\mathfrak{I}}(\mathsf{a}).$$

We proceed now to reason about the ϕ_i, assuming that ϕ_i is satisfiable, then $\exists \tau : \tau \vDash \phi_i$. We know that for every obligation, there exists at least one event in the metric-time word satisfying each obligation without violating any prohibition from the set of prohibitions. formally:

$$\begin{aligned} \exists (a_1,t_1),\ldots,(a_n,t_n) \ : \ \forall i \in [1,n] \quad & (a_i,t_i) \vDash O^{\mathfrak{I}_i}(\mathsf{a_i}) \\ & \text{and } \exists j \leq |\tau| : \tau(j) = (a_i,t_i) \\ & \text{and } (a_i,t_i) \vDash \mathcal{F}. \end{aligned}$$

The number of these *necessary events* is bounded by the number of obligations in the clause, that is, by $\eta(\phi_i)$. Moreover, since these events are part of a timed word, they must occur at distinct timestamps. Therefore, one can construct a timed word $\tau' \vDash \phi$ consisting solely of those events, with $|\tau'| = \eta(\phi_i)$. From Lemma 5, we also have that $\eta(\phi_i) \leq \eta(\phi)$, since ϕ_i is a clause in the *DNF* normalization of ϕ.

□

Theorem 1. *The satisfiability problem of* μMTNL *is* NP*-complete w.r.t the size of the formula* ϕ.

Proof sketch. We prove this result in two parts: (1) the satisfiability problem for μMTNL is in NP, and (2) it is NP-hard.

NP Membership: Given the bound $\eta(\phi)$, one can non-deterministically guess a candidate metric-time word τ of size at most $\eta(\phi)$ and verify whether it satisfies the formula in time polynomial with respect to $|\phi| \times \eta(\phi)$. Specifically, the verification step must, for each event in the metric-time word, check whether its timestamp falls within each of the intervals comprising the interval sets of the literals. Algorithm 1 presents the procedure checksat, which performs this verification recursively for formulas from μMTNL. For obligation and prohibition literals, the nested loops in the algorithm reflect this complexity: the formula size is determined by the number of intervals across all literals, while the size of the metric-time word is bounded by the number of obligation literals.

Algorithm 1: Procedure checksat

Input: A μMTNL formula ϕ and a metric-time word τ
Output: True if $\tau \vDash \phi$, False otherwise

```
switch φ do
    case O^𝔍(a) do
        foreach [t^i_min, t^i_max] ∈ 𝔍 do
            foreach (a_i, t_i) from τ do
                if a_i = a and t^i_min ≤ t_i ≤_∞ t^i_max then
                    return True
        return False
    case F^𝔍(a) do
        foreach [t^i_min, t^i_max] ∈ 𝔍 do
            foreach (a_i, t_i) from τ do
                if a_i = a and ≤ t_i ≤ then
                    return False
        return True
    case φ1 ⊓ φ2 do
        return checksat(φ1, τ) ∧ checksat(φ2, τ)
    case φ1 ⊔ φ2 do
        return checksat(φ1, τ) ∨ checksat(φ2, τ)
```

NP-hardness: We establish NP-hardness by reducing an arbitrary instance of the 3-SAT problem to an instance of μMTNL involving time-stamped obligations and prohibitions. The 3-SAT problem involves studying the satisfiability of a Boolean formula in CNF form, where each clause is formed by 3 literals. We suggest an encoding of the problem

into the logic as follows: each propositional literal is encoded as a literal from μMTNL on a generic and unique action symbol a , with distinct time points used to differentiate propositional variables. A positive literal (e.g., x_i) is represented as an obligation requiring a to occur at a specific time point (e.g., time i) to result on $O^{\{[i,i]\}}(\mathsf{a})$, while a negative literal (e.g., $\neg x_i$) is encoded as a prohibition of a at the same time point used for its corresponding positive literal, say $F^{\{[i,i]\}}(\mathsf{a})$. The logical structure of the original 3-SAT formula is preserved through the use of the $\sqcup$ (disjunction) and $\sqcap$ (conjunction) operators in μMTNL, thereby ensuring that satisfiability in the original formula corresponds to satisfiability in the constructed temporal instance.
For example, the 3-SAT formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee \neg x_1 \vee x_3) \wedge (\neg x_3 \vee x_2 \vee x_5)$ can be encoded as:

$$\left(\underbrace{O^{\{[1,1]\}}(\mathsf{a})}_{x_1} \sqcup \underbrace{F^{\{[2,2]\}}(\mathsf{a})}_{\neg x_2} \sqcup \underbrace{O^{\{[3,3]\}}(\mathsf{a})}_{x_3}\right) \sqcap \left(\underbrace{O^{\{[4,4]\}}(\mathsf{a})}_{x_4} \sqcup \underbrace{F^{\{[1,1]\}}(\mathsf{a})}_{\neg x_1} \sqcup \underbrace{O^{\{[3,3]\}}(\mathsf{a})}_{x_3}\right) \sqcap$$
$$\left(\underbrace{F^{\{[3,3]\}}(\mathsf{a})}_{\neg x_3} \sqcup \underbrace{O^{\{[2,2]\}}(\mathsf{a})}_{x_2} \sqcup \underbrace{O^{\{[5,5]\}}(\mathsf{a})}_{x_5}\right).$$

This approach maintains the logical equivalence of the original formula and avoids variable name conflicts by encoding each literal at a distinct time point.

This encoding preserves the satisfiability structure of the original Boolean formula: the μMTNL formula is satisfiable if and only if the original 3-SAT instance is satisfiable. Moreover, the translation is computable in polynomial time and introduces no blow-up in the size of the resulting formula.

Therefore, by proving NP Membership and Hardness, we conclude that the satisfiability problem for μMTNL is NP-complete w.r.t. the size of the formula. □

Corollary 1. *Given a bound κ on the number of intervals in any formula ϕ from μMTNL, the satisfiability problem is* NP*-complete w.r.t to $\eta(\phi)$, assuming κ is polynomially bounded in $\eta(\phi)$.*

SMT encoding of the satisfiability problem

We encode the satisfiability problem of a formula from μMTNL on an instance of SMT solving using Linear Integer Arithmetic(LIA). Specifically, for every $\phi \in \mu$MTNL, we define $\zeta(\phi)$, such that ϕ is satisfiable if and only if $\zeta(\phi)$ is satisfiable. The mapping from ϕ to $\zeta(\phi)$ is decomposed into three parts.

Timed words as constrained arrays of events: We represent a candidate model τ_{min} for a formula ϕ as an array of events of a bounded number of events written $|\tau_{min}|$, where each event is a pair (action, timestamp). Actions are treated as symbols from a predefined set representing Σ, and timestamps are represented as integers. In order to make the array τ_{min} simulate a timed word, we enforce a strictly increasing time value condition in the following formula:

$$\mathsf{Disjoint}(\tau_{min}) := \forall i < |\tau_{min}| - 1 : \mathsf{timestamp}(\tau_{min}[i]) < \mathsf{timestamp}(\tau_{min}[i+1]).$$

Constraint extraction We define the function CstrC that recursively collects the constraints for a formula ϕ. Those constraints are expressed on the events forming the model τ_{min}.

Obligation An obligation is transformed into the existence of an event in the metric-time word such that its action matches the obligation action and its timestamp lies within one of the specified intervals.

Example 24. *The obligation* $O^{\{[9,16]\}}(\mathsf{dap})$ *is encoded as:*

$$\mathsf{CstrC}(O^{\{[9,16]\}}(\mathsf{dap})) = \exists i < |\tau_{min}| : \mathsf{action}(\tau_{min}[i]) = \mathsf{dap} \wedge \big(9 \leq \mathsf{timestamp}(\tau_{min}[i]) \leq 16\big).$$

Prohibition A prohibition is transformed into an implication: for all events in the array, if an event has the prohibited action, its timestamp must not lie within the set of intervals.

Example 25. *The prohibition* $F^{\{[2,4],[8,24]}(\mathsf{dag})$ *is encoded as:*

$$\begin{aligned}\mathsf{CstrC}(F^{\{[2,4],[8,24]}(\mathsf{dag})) = \forall i < |\tau_{min}| : \big((2 \leq \mathsf{timestamp}(\tau_{min}[i]) \leq 4 \\ \vee 8 \leq \mathsf{timestamp}(\tau_{min}[i]) \leq 24) \implies \mathsf{action}(\tau_{min}[i]) \neq \mathsf{dag}\big).\end{aligned}$$

Conjunction and Disjunction Conjunctions and disjunctions of formulas are encoded using the boolean operators $\wedge$ and $\vee$, respectively.

To ensure decidability and reducibility to QF-LIA, we introduce an upper bound on the size of τ_{min} based on the formula. Let $\eta(\phi)$ denote the number of obligation literals in the syntactic closure of ϕ. We prove an upper bound on the size of the array $|\tau_{min}|$:

Consequently, with the constraints on the array τ_{min} representing the timed word model, along with the constraints collected from the formula ϕ, the resulting SMT encoding $\zeta(\phi)$ for deciding the satisfiability of the formula ϕ is as follows:

$$\zeta(\phi) := \mathsf{CstrC}(\phi) \wedge \mathsf{Disjoint}(\tau_{min}) \wedge \big(|\tau_{min}| \leq \eta(\phi)\big).$$

Example 26. *The whole SMT encoding of the use case* $\mathsf{UC} := \mathsf{DC} \sqcap \mathsf{SR} \sqcap \mathsf{MA}$ *is as follows:*

$$
\begin{aligned}
\zeta(\mathsf{UC}) =& \mathsf{CstrC}(\mathsf{UC}) \wedge \mathsf{Disjoint}(\tau_{min}) \wedge \big(|\tau_{min}| \leq \eta(\mathsf{UC})\big) \\
=& \Big[\big(\exists i < |\tau_{min}| \ : \mathsf{action}(\tau_{min}[i]) = \mathsf{dap} \wedge \big(9 \leq \mathsf{timestamp}(\tau_{min}[i]) \leq 16\big) \\
& \vee \big(\exists i < 2 : \mathsf{action}(\tau_{min}[i]) = \mathsf{dag} \wedge \big(9 \leq \mathsf{timestamp}(\tau_{min}[i]) \leq 16\big)\Big] \\
& \wedge \Big[\forall i < |\tau_{min}| : (10 \leq \mathsf{timestamp}(\tau_{min}[i]) \leq 14 \implies \mathsf{action}(\tau_{min}[i]) \neq \mathsf{dap}\Big] \\
& \wedge \Big[\forall i \leq |\tau_{min}| : (2 \leq \mathsf{timestamp}(\tau_{min}[i]) \leq 4 \vee 8 \leq \mathsf{timestamp}(\tau_{min}[i]) \leq 24) \\
& \implies \mathsf{action}(\tau_{min}[i]) \neq \mathsf{dag}\Big] \\
& \wedge \Big[\forall i < |\tau_{min}| - 1 : \mathsf{timestamp}(\tau_{min}[i]) < \mathsf{timestamp}(\tau_{min}[i+1])\Big] \\
& \wedge \Big[|\tau_{min}| \leq 2\Big].
\end{aligned}
$$

We implemented a satisfiability checker for μMTNL formulas using the Python programming language and the Z3 SMT solver.[3] The implementation leverages the solver capabilities of the SMT tool and returns a model τ_{min} when the formula is satisfiable; otherwise, it concludes unsatisfiability.

Example 27. *Below, we show the results when applying our solver to the sub-formulas from our use case:*

Solver Examples

#Satisfiable example UC_1'
Enter a formula from μMTNL*:* O dag *{[9,16]} &* F dag *{[10,14]}*
Satisfiable metric-time word with minimum length: 1
Event 0: Action = dag*, Timestamp = 15*

#Unsatisfiable example UC_2'
Enter a formula from μMTNL*:* O dag *{[9,16]} &* F dap *{[2,4],[8, 24]}*
Unsatisfiable formula.

#Use case UC
Enter a formula from μMTNL*: (*O dag *{[9,16]} ||* O dap *{[9,16]}) &* F dag *{[10,14]} &* F dag *{[2,4],[8,24]}*
Satisfiable metric-time word with minimum length: 1
Event 0: Action = dag*, Timestamp = 15*

[3] https://github.com/khrrzkrm/FMMTNL_Solver/

Additional Unsatisfiability Semantic Rules

We next formalize the propagation behavior of unsatisfiability in composite formulas. In particular, the following lemma describes how the conjunction and disjunction operators interact with an unsatisfiable sub-formula:

Lemma 10 (Properties of $\sqcap$ and $\sqcup$ with an unsatisfiable formula)**.** *For all formulas ϕ and ϕ' from μMTNL, if $[\![\phi]\!] = \emptyset$, then:*

$$[\![\phi \sqcap \phi']\!] = \emptyset \quad \textit{and} \quad [\![\phi \sqcup \phi']\!] = [\![\phi']\!].$$

That is:

1. *if a formula ϕ is unsatisfiable, then any conjunction formed by ϕ is also unsatisfiable.*
2. *if a formula ϕ is unsatisfiable, the satisfiability of the entire expression $\phi \sqcup \phi'$ relies solely on ϕ formula.*

Proof. By Lemma 3: 1. Intersection Case ($\sqcap$):

$$[\![\phi \sqcap \phi']\!] = [\![\phi]\!] \cap [\![\phi']\!].$$

Since $[\![\phi]\!] = \emptyset$, the intersection with any set $[\![\phi']\!]$ results in an empty set:

$$\emptyset \cap [\![\phi']\!] = \emptyset.$$

Hence, $[\![\phi \sqcap \phi']\!] = \emptyset$. □
2. Disjunction case($\sqcup$):

$$[\![\phi \sqcup \phi']\!] = [\![\phi]\!] \cup [\![\phi']\!].$$

Again, since $[\![\phi]\!] = \emptyset$, we get:

$$\emptyset \cup [\![\phi']\!] = [\![\phi']\!].$$

Thus, $[\![\phi \sqcup \phi']\!] = [\![\phi']\!]$. □

The converse behavior is also true for disjunction: if at least one sub-formula is satisfiable, then the disjunction is satisfiable. This is stated formally in the next lemma:

Lemma 11 (Disjunction and satisfiable sub-formula)**.** *For any two formulas ϕ and ϕ' from μMTNL: if ϕ is satisfiable then $\phi \sqcup \phi'$ is satisfiable.*

Proof. We have by Lemma 3 : $[\![\phi \sqcup \phi']\!] \equiv [\![\phi]\!] \cup [\![\phi']\!]$.
Assume ϕ is satisfiable, then $[\![\phi]\!] \neq \emptyset$.
By the definition of union, we have:
$[\![\phi]\!] \subseteq ([\![\phi]\!] \cup [\![\phi']\!])$, thus by the assumption we obtain: $([\![\phi]\!] \cup [\![\phi']\!]) \neq \emptyset$.
We conclude $[\![\phi \sqcup \phi']\!] \neq \emptyset$, which mean that if ϕ is satisfiable then $\phi \sqcup \phi'$ is satisfiable □

While satisfiability checking determines whether a normative specification admits at least one compliant behavior, it falls short in providing explanatory insight. Specifically, it does not reveal which parts of the specification are the root cause for unsatisfiability, nor does it clarify how or why certain norms render the system infeasible. It offers a binary answer—satisfiable or not—without tracing the origins of unsatisfiability or identifying potential resolutions. To overcome this limitation, we now turn to a more detailed analysis of normative conflicts, grounded in the semantics of μMTNL.

3.4 Syntactic Timed Normative Conflict Definition and Detection

The previous sections established the semantics of μMTNL, including the representation of obligations and prohibitions over interval sets and the construction of normal forms. We now address how to capture *conflicts* in temporal-deontic specifications formally.

Our motivation, grounded in practical use cases, is that deontic conflicts materialize at *specific time points* that must be isolated and explained. The goal is not only to show that a specification is inconsistent, but to determine *where*, *when*, and *why* inconsistency arises. We first define punctual ontic and deontic conflicts as atomic patterns. We then explain why these patterns do not suffice on their own, introduce syntax-aware invariants (LUCO and LAP) that stabilize conflict detection under rewriting, and finally develop a systematic procedure and quantitative measure that detect and classify all punctual conflicts that a formula *has*.

3.4.1 Syntactic Definitions for Conflicting Formulas

To reason about conflicts in μMTNL, we first need a precise vocabulary that identifies what constitutes a conflict and when a formula specifies a conflict. We therefore begin by isolating the most elementary patterns that arise at individual time points. These punctual patterns form the atomic units from which all further reasoning about conflicts in formulas *has or specifies*.

Punctual Conflicts: the Atomic Patterns

From a dilemma-oriented perspective, two types of timed normative conflicts arise. First, an agent may be simultaneously obliged and prohibited to perform the same action at the same time. Second, an agent may be obliged to perform two different actions at exactly the same time, while the agent can realize at most one of them. We formalize these punctual conflict patterns next.

Definition 24 (Punctual Conflicts)**.** *Let $a, b \in \Sigma$ and $t \in \mathbb{N}$.*

- *A* punctual deontic conflict *(**P-Deon-Conf**) occurs when an obligation and a prohibition prescribe and forbid the same action at the same time:*

$$O^{\{[t,t]\}}(\mathsf{a}) \sqcap F^{\{[t,t]\}}(\mathsf{a}).$$

- *A* punctual ontic conflict *(**P-Ont-Conf**) occurs when two distinct punctual obligations require different actions at the same time:*

$$O^{\{[t,t]\}}(\mathsf{a}) \sqcap O^{\{[t,t]\}}(\mathsf{b}).$$

These patterns are purely syntactic formulas of μMTNL. Whenever they appear as subterms of a product term composed of punctual literals (modulo commutativity and associativity), a conflict is present at that time point. A formula may contain one or several such conflicts as subformulas, and a single literal may participate in several conflicts.

Example 28 (Punctual conflicts appear as sub-formulas)**.** *We illustrate three formulas where conflicts occur as sub-terms. Underlined sub-terms indicate punctual conflicts.*

$\phi_1 := \Big(\underline{O^{\{[2,2]\}}(\mathsf{dap}) \sqcap F^{\{[2,2]\}}(\mathsf{dap})} \sqcap O^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[2,2]\}}(\mathsf{pick\_up})\Big) \sqcup O^{\{[0,7]\}}(\mathsf{dag})$
ϕ_1 contains a punctual deontic conflict on dap *at $t = 2$.*

$\phi_2 := \Big(\underline{O^{\{[5,5]\}}(\mathsf{dap}) \sqcap O^{\{[5,5]\}}(\mathsf{dag})}\Big) \sqcup \Big(\underline{O^{\{[5,5]\}}(\mathsf{dap}) \sqcap F^{\{[5,5]\}}(\mathsf{dap})}\Big) \sqcup O^{\{[0,6]\}}(\mathsf{pick\_up})$

Here, ϕ_2 contains two punctual conflicts at $t = 5$: one ontic and one deontic.

$\phi_3 := \Big(\underline{O^{\{[2,2]\}}(\mathsf{dap})} \sqcap \underline{O^{\{[2,2]\}}(\mathsf{dag})} \sqcap O^{\{[2,2]\}}(\mathsf{pick\_up})\Big) \sqcup \Big(O^{\{[2,2]\}}(\mathsf{dap}) \sqcap \underline{F^{\{[2,2]\}}(\mathsf{pick\_up})}\Big)$

ϕ_3 contains three punctual ontic conflicts at $t = 2$ between $\{\mathsf{dag}, \mathsf{dap}\}$, $\{\mathsf{dap}, \mathsf{pick\_up}\}$, and $\{\mathsf{dag}, \mathsf{pick\_up}\}$. Removing $O^{\{[2,2]\}}(\mathsf{dap})$ eliminates all conflicts because this literal participates in each of them.

Why Punctual Conflicts Are Not Enough

Real specifications rarely contain explicit punctual literals. Interval literals may conceal conflicts that only become visible after decomposition. For example, $F^{\{[1,1]\}}(\mathsf{a})$ is atomic, whereas $F^{\{[1,3]\}}(\mathsf{a})$ decomposes via Literal Compression:

$$F^{\{[1,3]\}}(\mathsf{a}) \equiv F^{\{[1,1]\}}(\mathsf{a}) \sqcap F^{\{[2,3]\}}(\mathsf{a}).$$

Thus, the punctual conflict $F^{\{[1,1]\}}(\mathsf{a}) \sqcap O^{\{[1,1]\}}(\mathsf{a})$ is visible in the decomposed form but *not* in $F^{\{[1,3]\}}(\mathsf{a}) \sqcap O^{\{[1,1]\}}(\mathsf{a})$.

Another case arises when conflicting literals do appear in the formula, yet their interaction is hidden by disjunction and does not manifest as a single product term. For instance,

$$(O^{\{[1,1]\}}(\mathsf{a}) \sqcup O^{\{[2,3]\}}(\mathsf{a})) \sqcap F^{\{[1,1]\}}(\mathsf{a}).$$

By distributivity,

$$(O^{\{[1,1]\}}(\mathsf{a}) \sqcup O^{\{[2,3]\}}(\mathsf{a})) \sqcap F^{\{[1,1]\}}(\mathsf{a}) \equiv (O^{\{[1,1]\}}(\mathsf{a}) \sqcap F^{\{[1,1]\}}(\mathsf{a})) \sqcup (O^{\{[2,3]\}}(\mathsf{a}) \sqcap F^{\{[1,1]\}}(\mathsf{a})).$$

The first disjunct exposes a punctual conflict at $t = 1$, while the second does not. This example shows that the conflict is not syntactically evident in the original formula but only appears after applying distributive expansion.

Hence, identifying conflicts cannot rely solely on the surface structure of a formula: conflicts may be latent within interval composition or disjunctive structure and require specific *rewriting* to be made explicit.

Why Naive Equivalence-Based Detection Fails

A first idea would be to declare that a formula "has a conflict" whenever it is semantically equivalent to another formula that visibly contains a punctual conflict as defined in [KSL24]. This criterion is too weak: Boolean manipulations can erase genuine conflicts or introduce spurious ones.

Example 29 (Illustration of problems with naive equivalence-based detection)**.** *We demonstrate equivalences by erasing or introducing conflicts using the absorption properties of disjunction of an unsatisfiable formula with any other formula, as shown in Lemma .10.*
(i) ***Erasing a genuine conflict.***

$$\phi := (O^{\{[2,2]\}}(\mathsf{a}) \sqcap F^{\{[2,2]\}}(\mathsf{a})) \sqcup \phi'$$

is equivalent to ϕ', so naive equivalence would falsely conclude that ϕ has no conflict.

(ii) ***Creating a spurious conflict.***

$$\phi' := \phi \sqcup (O^{\{[3,3]\}}(\mathsf{a}) \sqcap F^{\{[3,3]\}}(\mathsf{a}))$$

is equivalent to ϕ, but naive equivalence would now falsely conclude that ϕ has a conflict.

To avoid such pathologies, conflict extraction must respect the syntactic structure of the formula.

Syntax-Aware Conflict Detection: LUCO and LAP

We therefore introduce two invariants that govern how literal refinements behave in conflict detection: (i) *Lowest Upper Common Operator (LUCO):* preserves the Boolean position where literals meet; and (ii) *Literal Ancestor Preservation (LAP):* ensures punctual descendants refine—but never alter—their ancestor literals.

These invariants ensure that conflicts cannot vanish or appear spuriously during normalization.

Definition 25 (Lowest upper common Boolean operator under unique occurrence)**.** *Let $\phi \in \mu\mathsf{MTNL}$ and let $\mathsf{T}(\phi)$ be its Boolean syntax tree whose internal nodes are $\{\sqcap, \sqcup\}$ and whose leaves are literals. Assume ℓ_i, ℓ_j each occur exactly once in ϕ. The* lowest upper common Boolean operator *(*LUCO*) of ℓ_i and ℓ_j in ϕ, denoted $\mathrm{LUCO}_\phi(\ell_i, \ell_j)$, is the unique node v of $\mathsf{T}(\phi)$ such that:*

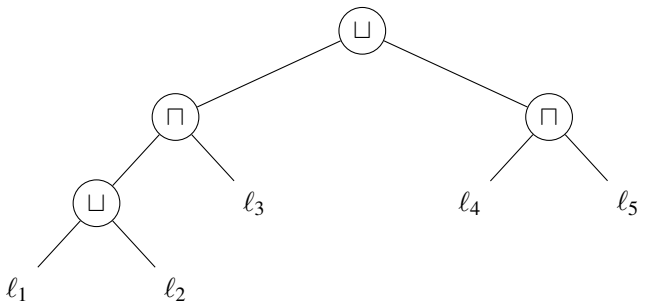
(i) v is labeled by $\sqcap$ or $\sqcup$;

(ii) v lies on both root-to-ℓ_i and root-to-ℓ_j paths; and

(iii) no proper descendant of v satisfies (i) *and* (ii)*.*

We write $\mathrm{LUCO}_\phi(\ell_i, \ell_j) = \sqcap$ (resp. $= \sqcup$) to denote its label.

Example 30 (LUCO on a formula with 5 literals)**.** *Let $\phi := \Big((\ell_1 \sqcup \ell_2) \sqcap \ell_3\Big) \sqcup \Big(\ell_4 \sqcap \ell_5\Big)$, with unique occurrences of each literal. Then:* ***Syntax tree of ϕ,*** *$\mathsf{T}(\phi)$.*



$$\mathrm{LUCO}_\phi(\ell_1,\ell_2)=\sqcup,\quad \mathrm{LUCO}_\phi(\ell_1,\ell_3)=\mathrm{LUCO}_\phi(\ell_2,\ell_3)=\sqcap,\quad \mathrm{LUCO}_\phi(\ell_4,\ell_5)=\sqcap,$$

and for any $x\in\{\ell_1,\ell_2,\ell_3\}$ *and* $y\in\{\ell_4,\ell_5\}$,

$$\mathrm{LUCO}_\phi(x,y)=\sqcup \quad \textit{(the root).}$$

As literals in our logic are not atomic, we introduce the notion of ancestor preservation with respect to time points in interval sets. Informally, this invariant ensures that normalization only refines a literal: no new time points are introduced beyond those originally specified, and none of the initial ones are lost. The modality and action remain unchanged, and the resulting punctual descendants, when recombined through their LUCO, reconstruct exactly the original literal.

Definition 26 (Literal ancestor preservation (LAP))**.** *Let* ϕ *and* ϕ' *be two formula from* μMTNL. *We say that* ϕ' *satisfies* literal ancestor preservation *of* ϕ *if there exists a mapping between literals in* ϕ *and literals in* ϕ' *such that:*

(i) (***Downward preservation***) *For every literal* ℓ' *in* ϕ', *there exists an ancestor literal* ℓ *in* ϕ *with the same modality and action, and whose interval set contains that of* ℓ'.

(ii) (***Upward coverage***) *For every literal* ℓ *in* ϕ, *the set of its descendants* $\{\ell'_1,\ell'_2,\ldots,\ell'_n\}$ *in* ϕ' *(i.e. those having* ℓ *as ancestor) combine under the Boolean operator determined by their Lowest upper common Boolean operator in* ϕ' *to a sub-formula that is syntactically equivalent to* ℓ.

These invariants ensure that literals and their Boolean combinations are neither added nor altered during the successive normalizations, which is precisely the guarantee required to prove correctness.

A Stable Definition of "Has a Conflict"

We now define the correct notion of when a formula *has* a conflict.

Definition 27 (Formula has a conflict)**.** *A formula* ϕ has a conflict *iff there exists a formula* ϕ' *such that:*

(i) $\phi'\equiv\phi$,

(ii) ϕ' *preserves* LUCO *and* LAP *with respect to* ϕ,

(iii) ϕ' *has a punctual conflict as a sub-formula.*

Example 31 (Preliminary exhibition of conflicts)**.**

Let $\phi := O^{\{[2,3]\}}(\mathsf{a}) \sqcap F^{\{[2,4]\}}(\mathsf{a})$

and

$$\phi' := \Big(O^{\{[2,2]\}}(\mathsf{a}) \sqcap F^{\{[2,2]\}}(\mathsf{a}) \sqcap F^{\{[3,4]\}}(\mathsf{a})\Big) \sqcup \Big(O^{\{[3,3]\}}(\mathsf{a}) \sqcap F^{\{[2,4]\}}(\mathsf{a})\Big).$$

We justify that ϕ *specifies the conflict* $O^{\{[2,2]\}}(\mathsf{a}) \sqcap F^{\{[2,2]\}}(\mathsf{a})$ *as follows:*

(i) ***Semantic equivalence.*** *By construction,* ϕ' *is simply the punctual decomposition of* ϕ*:*

$$O^{\{[2,3]\}}(\mathsf{a}) \equiv O^{\{[2,2]\}}(\mathsf{a}) \sqcup O^{\{[3,3]\}}(\mathsf{a}), \qquad F^{\{[2,4]\}}(\mathsf{a}) \equiv F^{\{[2,2]\}}(\mathsf{a}) \sqcap F^{\{[3,4]\}}(\mathsf{a}).$$

Substituting these equivalences yields exactly ϕ'*, hence* $\phi \equiv \phi'$*.*

(ii) LUCO ***preservation.*** *In* ϕ*, the two literals* $O^{\{[2,3]\}}(\mathsf{a})$ *and* $F^{\{[2,4]\}}(\mathsf{a})$ *occur under a conjunctive node* $\sqcap$*. In* ϕ'*, their punctual descendants (e.g.* $O^{\{[2,2]\}}(\mathsf{a})$ *and* $F^{\{[2,2]\}}(\mathsf{a})$*) also occur under a conjunctive* $\sqcap$ *inside each product term. Thus,* LUCO *is preserved.*

(iii) LAP ***preservation.*** *Each punctual literal in* ϕ' *has a corresponding ancestor in* ϕ*:*

- $O^{\{[2,2]\}}(\mathsf{a})$ *and* $O^{\{[3,3]\}}(\mathsf{a})$ *are descendants of* $O^{\{[2,3]\}}(\mathsf{a})$*, and their interval sets are contained in* $[2,3]$*.*
- $F^{\{[2,2]\}}(\mathsf{a})$ *and* $F^{\{[3,4]\}}(\mathsf{a})$ *are descendants of* $F^{\{[2,4]\}}(\mathsf{a})$*, and their interval sets are contained in* $[2,4]$*.*

Moreover, the descendants recombine conjunctively or disjunctively exactly as required to recover their ancestors. Hence LAP *is satisfied.*

(iv) ***Conflicts appear as subterm.*** *The first disjunct of* ϕ' *visibly contains the punctual conflict*

$$O^{\{[2,2]\}}(\mathsf{a}) \sqcap F^{\{[2,2]\}}(\mathsf{a}).$$

Therefore, ϕ has *a conflict according to Definition 27, witnessed by* ϕ'*.*

This definition stabilizes the notion of conflict under normalization: conflicts cannot disappear or be fabricated through Boolean transformations, and later sections will show that every conflict that a formula has can be made explicit in its *DPNF* form.

The definition requires only that a formula have at least one punctual conflict, although many different rewritings ϕ' may preserve LUCO and LAP while exposing one or several such conflicts. In the next section, we develop an algorithm that uncovers *all* conflicts that a formula has according to this definition.

3.4.2 Systematic Enumeration of all Punctual Conflicts

To exhibit such conflicts systematically, we define the *disjunctive punctual normal form* (*DPNF*). In this form, each disjunct corresponds to a punctual product term, and conflicts can be localized by analyzing these terms individually. We prove that transformation into *DPNF* preserves semantics, and we establish soundness and completeness of the conflict-detection procedure (Algorithm 2) with respect to the definition above. This ensures that every punctual conflict present in a formula can be effectively detected, and that no spurious conflicts are introduced.

Beyond detection, we introduce the measure of *conflict density* to quantify the proportion of conflicting punctual terms in a formula. Formally, conflict density is defined as the ratio between the number of detected punctual conflicts and the total number of disjunctive product terms in *DPNF*. A high density reflects an increased level of normative inconsistency, reducing clarity for agents seeking to comply with the norms.

We now start by defining a new normal form specific to our logic in order to exhibit punctual conflict by decomposing literals into punctual literals.

Punctual Normal Form

We want to syntactically rewrite any literal from a formula to a combination of punctual literals. Any non-punctual interval $I = [x,y]$ with $x \neq y$ can be decomposed as

$$I = [x,x] \cup [x{+}1,y],$$

and, by orienting the equivalences **ObDis** and **FConj**, we obtain the following rewriting rules:

$$\frac{x \neq y}{O^{(\{[x,y]\}\cup\mathfrak{I})}(\mathsf{a}) \to O^{\{[x,x]\}}(\mathsf{a}) \sqcup O^{(\{[x+1,y]\}\cup\mathfrak{I})}(\mathsf{a})} \textbf{(RightPunctOb)}$$

$$\frac{x \neq y}{F^{(\{[x,y]\}\cup\mathfrak{I})}(\mathsf{a}) \to F^{\{[x,x]\}}(\mathsf{a}) \sqcap F^{(\{[x+1,y]\}\cup\mathfrak{I})}(\mathsf{a})} \textbf{(RightPunctF)}$$

Figure 3.2: Literal-to-right punctual decomposition rules

Example 32. *For* $O^{\{[2,4],[6,8]\}}(\mathsf{a})$*, applying* ***RightPunctOb*** *yields*

$$O^{\{[2,4],[6,8]\}}(\mathsf{a}) \to O^{\{[2,2]\}}(\mathsf{a}) \sqcup O^{\{[3,4],[6,8]\}}(\mathsf{a}),$$

introducing the punctual literal $O^{\{[2,2]\}}(\mathsf{a})$.

Repeated application of these rules unrolls non-punctual intervals until their upper bounds. However, prohibitions may include infinite intervals (e.g. $[k,\infty]$), making unrestricted expansion non-terminating. To ensure finiteness, we define a bounded variant.

Definition 28 (Punctual Normal Form up to k)**.** *Let $k \in \mathbb{N}$. A formula ϕ is in* punctual normal form up to k *if every literal in ϕ is either:*

- *a punctual literal $O^{\{[t,t]\}}(\mathsf{a})$ or $F^{\{[t,t]\}}(\mathsf{a})$ with $t \leq k$, or*
- *a literal $O^{\mathfrak{I}}(\mathsf{a})$ or $F^{\mathfrak{I}}(\mathsf{a})$ such that $\min(\mathfrak{I}) > k$.*

We now guard the previous rules by a cutoff point k, as shown below.

$$\frac{x \neq y \ \wedge \ x \leq k}{O^{(\{[x,y]\}\cup\mathfrak{I})}(\mathsf{a}) \to O^{\{[x,x]\}}(\mathsf{a}) \sqcup O^{(\{[x+1,y]\}\cup\mathfrak{I})}(\mathsf{a})} \ \textbf{(RightPunctOb}^k\textbf{)}$$

$$\frac{x \neq y \ \wedge \ x \leq k}{F^{(\{[x,y]\}\cup\mathfrak{I})}(\mathsf{a}) \to F^{\{[x,x]\}}(\mathsf{a}) \sqcap F^{(\{[x+1,y]\}\cup\mathfrak{I})}(\mathsf{a})} \ \textbf{(RightPunctF}^k\textbf{)}$$

Figure 3.3: Literal-to-right punctual decomposition rules up to a finite bound k

Example 33 (Applying **RightPunctF**^{k} to $F^{\{[2,4],[8,24]\}}(\mathsf{dag})$ with $k = 16$)**.**

$$\begin{aligned}\ell &:= F^{\{[2,4],[8,24]\}}(\mathsf{dag}) \\ &\to F^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[3,4],[8,24]\}}(\mathsf{dag}) \\ &\to F^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[3,3]\}}(\mathsf{dag}) \sqcap \cdots \sqcap F^{\{[16,16]\}}(\mathsf{dag}) \sqcap F^{\{[17,24]\}}(\mathsf{dag}).\end{aligned}$$

No further rule applies beyond $k = 16$, yielding the punctual normal form up to the bound.

Definition 29 (Transformer to Punctual Normal Form up to k)**.** *Let $\phi \in \mu\mathsf{MTNL}$ and $k \in \mathbb{N}$. The recursive function $PNF(\phi, k)$ computes the punctual normal form of ϕ up to k:*

$$PNF(\phi,k) := \begin{cases} \bigsqcup_{t\in\mathfrak{I}\ominus[k+1,\infty]} O^{\{[t,t]\}}(\mathsf{a}) \sqcup O^{\mathfrak{I}\cap[k+1,\infty]}(\mathsf{a}) & \text{if } \phi = O^{\mathfrak{I}}(\mathsf{a}), \\ \bigsqcap_{t\in\mathfrak{I}\ominus[k+1,\infty]} F^{\{[t,t]\}}(\mathsf{a}) \sqcap F^{\mathfrak{I}\cap[k+1,\infty]}(\mathsf{a}) & \text{if } \phi = F^{\mathfrak{I}}(\mathsf{a}), \\ PNF(\phi_1,k) \sqcup PNF(\phi_2,k) & \text{if } \phi = \phi_1 \sqcup \phi_2, \\ PNF(\phi_1,k) \sqcap PNF(\phi_2,k) & \text{if } \phi = \phi_1 \sqcap \phi_2. \end{cases}$$

Here $\mathfrak{I} \ominus [k+1,\infty]$ restricts $\mathfrak{I}$ to time points $\leq k$.

Example 34 (*PNF*($\mathsf{UC}_1', 16$))**.** *For*

$$\mathsf{UC}_1' := O^{\{[9,16]\}}(\mathsf{dag}) \sqcap F^{\{[2,4],[8,24]\}}(\mathsf{dag}) \sqcap F^{\{[10,14]\}}(\mathsf{dap}),$$

we obtain:

$$PNF(\mathsf{UC}_1', 16) = \left(\bigsqcup_{t=9}^{16} O^{\{[t,t]\}}(\mathsf{dag})\right) \sqcap \left(\prod_{t=10}^{14} F^{\{[t,t]\}}(\mathsf{dap})\right) \sqcap \left(\prod_{t=2}^{4} F^{\{[t,t]\}}(\mathsf{dag}) \sqcap \prod_{t=8}^{16} F^{\{[t,t]\}}(\mathsf{dag}) \sqcap F^{\{[17,24]\}}(\mathsf{dag})\right).$$

Intervals exceeding the bound remain non decomposed, ensuring termination.

We now compare the syntactic size of a formula with that of its punctual normal form.

Lemma 12 (Size Blow-Up of Literals under *PNF*)**.** *Let $\ell = D^{\mathfrak{I}}(\mathsf{a})$ be a literal of $\mu\mathsf{MTNL}$, where $D \in \{\mathsf{O}, \mathsf{F}\}$, and let $k \in \mathbb{N}$ be a finite bound. Then:*

$$\frac{|PNF(\ell, k)|}{|\ell|} \leq 5\,\iota(\mathfrak{I}) - \frac{1}{card(\mathfrak{I})},$$

where $\iota(\mathfrak{I}) := \frac{|\mathfrak{I}|}{card(\mathfrak{I})}$ is the expansion factor *of the interval set $\mathfrak{I}$.*

Proof. For a literal $\ell = D^{\mathfrak{I}}(\mathsf{a})$,

$$|\ell| = 2 + 2card(\mathfrak{I}),$$

counting the operator, action, and the two numeric bounds per interval.

Under *PNF*, each time point t in $\mathfrak{I} \ominus [k+1, \infty]$ becomes a punctual literal $D^{\{[t,t]\}}(\mathsf{a})$, connected by conjunctions or disjunctions depending on D. The number of punctual literals is at most $|\mathfrak{I}|$, and the number of binary connectives is $|\mathfrak{I}| - 1$.

Each punctual literal contributes a fixed size of 4, giving

$$|PNF(\ell, k)| \leq 4|\mathfrak{I}| + (|\mathfrak{I}| - 1) + C = 5|\mathfrak{I}| - 1 + C,$$

where C accounts for a possible tail term and is absorbed in the bound.

Hence,

$$\frac{|PNF(\ell, k)|}{|\ell|} \leq \frac{5|\mathfrak{I}| - 1}{2 + 2card(\mathfrak{I})} = \frac{5\iota(\mathfrak{I})card(\mathfrak{I}) - 1}{2 + 2card(\mathfrak{I})} \leq 5\,\iota(\mathfrak{I}) - \frac{1}{card(\mathfrak{I})}.$$

□

Lemma 13 (Formula Size Blow-Up and Average Expansion Factor)**.** *Let ϕ be a formula in μMTNL and $k \in \mathbb{N}$ a finite bound. Let ϕ_x be the set of literals of ϕ, and $\mathcal{I}_\phi$ the set of interval sets occurring in those literals. Define the* average expansion factor *of ϕ as:*

$$\iota_a(\phi) := \frac{1}{\mathit{card}(\mathcal{I}_\phi)} \sum_{\mathfrak{I} \in \mathcal{I}_\phi} \iota(\mathfrak{I}).$$

Then the size of $PNF(\phi,k)$ grows linearly with both $\iota_a(\phi)$ and $|\phi|$:

$$|PNF(\phi,k)| = \mathcal{O}\big(\iota_a(\phi) \cdot |\phi|\big).$$

A *DPNF* Algorithm for Enumerating Punctual Conflicts

To systematically expose all punctual conflicts specified by a formula, we exploit the punctual and disjunctive normal forms introduced earlier. Given $\phi \in \mu$MTNL, we first transform it into $PNF(\phi,k)$, where k is the maximal time point appearing in any obligation literal of ϕ. Next, we convert $PNF(\phi,k)$ into its disjunctive normal form *DNF*, producing a disjunction of product terms in which punctual conflicts can be directly pattern-matched. Algorithm 2 implements this procedure.

Algorithm 2: Procedure PUNCTUALCONFLICTS $\mathcal{PC}$

Input: A formula $\phi \in \mu$MTNL

Output: A set $\mathcal{PC}$ of tuples $(\mathsf{pt},\mathcal{C})$ where pt is a product term and $\mathcal{C}$ is a set of punctual conflicts in pt

$k \leftarrow \textsc{MaxOPoint}(\phi)$; `// Max obligation time point in` ϕ
$\phi_{\text{pnf}} \leftarrow PNF(\phi,k)$; `// Punctual normal form up to` k
$\phi_{\text{dnf}} \leftarrow DNF(\phi_{\text{pnf}})$; `// Convert to disjunctive normal form`
$\mathcal{PC} \leftarrow \emptyset$; `// Initialize conflict set`
foreach *product term* pt *in* ϕ_{dnf} **do**
  $\mathcal{C} \leftarrow \emptyset$; `// Initialize local conflict set`
  foreach *pair of literals* (ℓ_1,ℓ_2) *in* pt **do**
    if (ℓ_1,ℓ_2) *matches* ***P-Deon-Conf*** *or* ***P-Ont-Conf*** **then**
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{\ell_1 \sqcap \ell_2\}$; `// Add conflict`
  if $\mathcal{C} \neq \emptyset$ **then**
    $\mathcal{PC} \leftarrow \mathcal{PC} \cup \{(\mathsf{pt},\mathcal{C})\}$; `// Add conflict tuple`
return $\mathcal{PC}$

Here, $\textsc{MaxOPoint}(\phi)$ returns the largest finite time point from any obligation interval in ϕ. The algorithm then scans each product term of the disjunctive form and records all literal pairs matching the deontic or ontic conflict patterns, returning them as tuples $(\mathsf{pt},\mathcal{C})$.

Step	Value	Size
ϕ	$O^{\{[1,2]\}}(\mathsf{dag}) \sqcap F^{\{[2,5]\}}(\mathsf{dag}) \sqcap O^{\{[1,2]\}}(\mathsf{pick\_up})$	14
$PNF(\phi,2)$	$\Big(O^{\{[1,1]\}}(\mathsf{dag}) \sqcup O^{\{[2,2]\}}(\mathsf{dag})\Big) \sqcap \Big(F^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[3,5]\}}(\mathsf{dag})\Big) \sqcap \Big(O^{\{[1,1]\}}(\mathsf{pick\_up}) \sqcup O^{\{[2,2]\}}(\mathsf{pick\_up})\Big)$	33
$DPNF(\phi,2))$	$\Big(O^{\{[1,1]\}}(\mathsf{dag}) \sqcap F^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[3,5]\}}(\mathsf{dag}) \sqcap O^{\{[1,1]\}}(\mathsf{pick\_up})\Big)\ \underline{pt_1}$ $\sqcup \Big(O^{\{[1,1]\}}(\mathsf{dag}) \sqcap F^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[3,5]\}}(\mathsf{dag}) \sqcap O^{\{[2,2]\}}(\mathsf{pick\_up})\Big)\ \underline{pt_2}$ $\sqcup \Big(O^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[3,5]\}}(\mathsf{dag}) \sqcap O^{\{[1,1]\}}(\mathsf{pick\_up})\Big)\ \underline{pt_3}$ $\sqcup \Big(O^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[3,5]\}}(\mathsf{dag}) \sqcap O^{\{[2,2]\}}(\mathsf{pick\_up})\Big)\ \underline{pt_4}$	76
$\mathcal{PC}(\phi)$	$\big(\mathsf{pt}_1, \{O^{\{[1,1]\}}(\mathsf{dag}) \sqcap O^{\{[1,1]\}}(\mathsf{pick\_up})\}\big)$, $\big(\mathsf{pt}_3, \{O^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[2,2]\}}(\mathsf{dag})\}\big)$ $\big(\mathsf{pt}_4, \{O^{\{[2,2]\}}(\mathsf{dag}) \sqcap F^{\{[2,2]\}}(\mathsf{dag}), O^{\{[2,2]\}}(\mathsf{dag}) \sqcap O^{\{[2,2]\}}(\mathsf{pick\_up})\}\big)$	–

Table 3.4: Normal form analysis with sizes for $\phi := O^{\{[1,2]\}}(\mathsf{dag}) \sqcap F^{\{[2,5]\}}(\mathsf{dag}) \sqcap O^{\{[1,2]\}}(\mathsf{pick\_up})$.

Example 35 (Conflict detection and density)**.** *To illustrate, consider a compact specification combining both deontic and ontic punctual conflicts:*

$$\phi := O^{\{[1,2]\}}(\mathsf{dag}) \sqcap F^{\{[2,5]\}}(\mathsf{dag}) \sqcap O^{\{[1,2]\}}(\mathsf{pick\_up}).$$

The maximal obligation time point is $k = 2$. Applying $PNF(\phi,2)$ expands obligations and prohibitions up to $t \leq 2$, while retaining the tail $[3,5]$. Subsequent DNF and conflict extraction yield the results summarized in Table Table 3.4.

Three punctual conflicts are found: (1) an ontic conflict at $t = 1$ between $O^{\{[1,1]\}}(\mathsf{dag})$ and $O^{\{[1,1]\}}(\mathsf{pick\_up})$; (2) a deontic conflict at $t = 2$ between $O^{\{[2,2]\}}(\mathsf{dag})$ and $F^{\{[2,2]\}}(\mathsf{dag})$; (3) a mixed case at pt_4 containing both.

Lemma 14 (*DPNF* preserves semantics)**.** *For any $\phi \in \mu\mathsf{MTNL}$ and $k \in \mathbb{N}$,*

$$[\![\phi]\!] = [\![DNF(PNF(\phi,k))]\!], \qquad \textit{i.e.} \quad DNF(PNF(\phi,k)) \equiv \phi.$$

Proof. By Lemma 6, $PNF(\phi,k) \equiv \phi$. By Lemma 5(i), $DNF(\phi) \equiv \phi$. Composing these yields $DNF(PNF(\phi,k)) \equiv \phi$. □

Before proving correctness, we establish the *structural invariants* preserved by the punctual and disjunctive normalizations. These invariants ensure that the transformations *PNF*

and *DNF* maintain the Boolean hierarchy and the syntactic relationships between literals in ϕ. In particular:

(i) the lowest upper common Boolean operator (LUCO) between literals is preserved;

(ii) ancestor relations between literals are maintained; and

(iii) punctual descendants are introduced exactly at overlapping time points.

Together, these guarantees allow the algorithm to detect every punctual conflict without creating spurious ones.

Lemma 15 (LUCO preservation under *PNF*)**.** *Let $\phi \in \mu\mathsf{MTNL}$, $k \in \mathbb{N}$, and assume unique occurrences of literals in ϕ. For any literals ℓ, ℓ' in ϕ and their punctual (or residual) descendants $\hat{\ell}, \hat{\ell}'$ in PNF(ϕ, k), we have*

$$\mathrm{LUCO}_{PNF(\phi,k)}(\hat{\ell}, \hat{\ell}') = \mathrm{LUCO}_{\phi}(\ell, \ell').$$

Proof. By Definition 29, *PNF* acts as homomorphism on conjunction and disjunction. Transformations occur locally: each literal is unfolded *in place* into a disjunction (for *O*) or conjunction (for *F*) of punctual or residual components. No Boolean connective along the path from root to leaf is moved across levels. Hence the lowest upper common operator (LUCO) dominating both positions remains unchanged. □

Lemma 16 (Literal ancestor preservation under *PNF*)**.** *Let $\phi \in \mu\mathsf{MTNL}$ and $k \in \mathbb{N}$. Then PNF(ϕ, k) satisfies* literal ancestor preservation *of ϕ (Definition 26).*

Proof. (*Downward*) Each literal ℓ' produced by *PNF*(ϕ, k) descends from a unique literal ℓ in ϕ with identical modality and action. Its interval set is contained in that of ℓ, corresponding to punctual points $t \leq k$ or the residual tail.

(*Upward*) For any literal ℓ in ϕ, its descendants in *PNF*(ϕ, k) are: (a) for *O*, the family $\{O^{\{[t,t]\}}(\mathsf{a}) \mid t \in \mathfrak{I} \cap [0,k]\}$, possibly plus $O^{\mathfrak{I} \cap [k+1,\infty]}(\mathsf{a})$; (b) for *F*, the analogous conjunctive family. Their nearest common Boolean operator in *PNF*(ϕ, k) is the one introduced locally at ℓ—a disjunction for *O* or conjunction for *F*, and by **ObDis/FConj** the resulting expression is syntactically equivalent to ℓ. □

Lemma 17 (Witness punctualization under *PNF*)**.** *Let $\phi \in \mu\mathsf{MTNL}$ with unique literal occurrences and $k = \textsc{MaxOPoint}(\phi)$. Suppose literals $\ell = D^{\mathfrak{I}}(\mathsf{a})$ and $\ell' = D'^{,\mathfrak{I}'}(\mathsf{a}')$ occur in ϕ with $\mathrm{LUCO}_{\phi}(\ell, \ell') = \sqcap$ and share some $t \in \mathfrak{I} \cap \mathfrak{I}'$ where $t \leq k$. Then PNF(ϕ, k) contains punctual descendants $\hat{\ell}^p = D^{\{[t,t]\}}(\mathsf{a})$ and $\hat{\ell}'^p = D'^{\{[t,t]\}}(\mathsf{a}')$ that remain under the same conjunctive* LUCO *relation.*

Proof. From Definition 29, $t \leq k$ and $t \in \mathfrak{I}, \mathfrak{I}'$ ensure creation of punctual descendants exactly at the positions of ℓ and ℓ'. By Lemma 15, the corresponding conjunctive LUCO is preserved. □

Lemma 18 (LUCO preservation and localization under *DNF*)**.** *Let $\phi \in \mu\mathsf{MTNL}$ with unique literal occurrences.*

(i) ***LUCO preservation.*** *For any literals ℓ, ℓ' in ϕ and their descendants $\hat{\ell}, \hat{\ell}'$ in DNF(ϕ) at corresponding positions,*

$$\mathrm{LUCO}_{DNF(\phi)}(\hat{\ell}, \hat{\ell}') = \mathrm{LUCO}_{\phi}(\ell, \ell').$$

(ii) ***Localization to product terms.*** *If $\mathrm{LUCO}_{\phi}(\ell, \ell') = \sqcap$, then some product term pt of DNF(ϕ) contains both $\hat{\ell}$ and $\hat{\ell}'$. If $\mathrm{LUCO}_{\phi}(\ell, \ell') = \sqcup$, no product term contains both.*

Proof. The *DNF* normalization is obtained by orienting distributivity (e.g. $\phi_1 \sqcap (\phi_2 \sqcup \phi_3) \to (\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3)$) Moreover, applying associativity and commutativity. These rewritings never invert $\sqcap/\sqcup$ but only propagate one operator outward, So dominance relations along root-to-leaf paths remain intact.

(i) Suppose, for contradiction, that a pair of literals changes its LUCO after normalization. Then their former lowest common operator either ceases to be common or lowest. Since *DNF* introduces no deeper operator and preserves labels, neither case can occur; hence LUCO is preserved.

(ii) Under the cross-product semantics of *DNF*, a conjunctive node yields product terms pairing one choice from each branch, so descendants under the same $\sqcap$ co-occur, while those under $\sqcup$ are separated. □

Example 36 (LUCO preservation under *DNF*)**.** *For the formula of Example 30,*

$$\phi = \big((\ell_1 \sqcup \ell_2) \sqcap \ell_3\big) \sqcup \big(\ell_4 \sqcap \ell_5\big), \qquad DNF(\phi) = (\ell_1 \sqcap \ell_3) \sqcup (\ell_2 \sqcap \ell_3) \sqcup (\ell_4 \sqcap \ell_5).$$

The LUCO *relations are preserved:*

$$\begin{aligned}
&\mathrm{LUCO}_{DNF(\phi)}(\ell_1, \ell_3) = \sqcap = \mathrm{LUCO}_{\phi}(\ell_1, \ell_3)\\
&\mathrm{LUCO}_{DNF(\phi)}(\ell_2, \ell_3) = \sqcap = \mathrm{LUCO}_{\phi}(\ell_2, \ell_3),\\
&\mathrm{LUCO}_{DNF(\phi)}(\ell_4, \ell_5) = \sqcap = \mathrm{LUCO}_{\phi}(\ell_4, \ell_5),\\
&\mathrm{LUCO}_{DNF(\phi)}(\ell_1, \ell_2) = \sqcup = \mathrm{LUCO}_{\phi}(\ell_1, \ell_2).
\end{aligned}$$

Thus, the LUCO *label is invariant, even though its syntactic position may shift.*

The preceding lemmas ensure that the normalization pipeline preserves both syntactic structure (LUCO) and lineage of literals (LAP). Hence, any punctual conflict specified in ϕ remains traceable in $DNF(PNF(\phi,k))$, allowing for sound and complete conflict extraction.

Theorem 2 (Correctness of Algorithm 2)**.** *Let ϕ be any formula in μ*MTNL*, and let $k = \text{MAXOPOINT}(\phi)$. Then Algorithm 2 is* sound and complete *for detecting punctual conflicts in ϕ, meaning the set $\mathcal{PC}(\phi)$ returned by the algorithm contains* exactly all *the punctual conflicts specified by ϕ.*

Proof. **(I) Soundness.** For any $(\text{pt}, \mathcal{C}) \in \mathcal{PC}(\phi)$ and $\hat{\ell} \sqcap \hat{\ell}' \in \mathcal{C}$, the literals $\hat{\ell}, \hat{\ell}'$ are punctual and match either **(P-Deon-Conf)** or **(P-Ont-Conf)** within some product term of $DNF(PNF(\phi,k))$. By Lemma 16, each has an ancestor ℓ, ℓ' in ϕ with the same modality and covering t. Lemmas 15 and 18 guarantee that their LUCO is preserved, so $\ell \sqcap \ell'$ is in fact specified in ϕ. Hence, no *fabricated* conflicts are introduced.

(II) Completeness.

(1) *All relevant time points exposed.* Choosing $k = \text{MAXOPOINT}(\phi)$ unfolds every point relevant to O/O or O/F conflicts. No conflict involves two prohibitions. By Lemmas 15 and 18, all LUCO relations are preserved, so no punctual literal is removed.

(2) *All conflicts localized and enumerated.* If the literals ℓ_i, ℓ_j form a conflict in ϕ with $\text{LUCO}_\phi(\ell_i, \ell_j) = \sqcap$ and shared t, then by Lemma 17 punctual descendants appear in $PNF(\phi,k)$ under the same LUCO. By Lemma 18(ii) they co-occur in a product term of $DNF(PNF(\phi,k))$. The algorithm checks all pairs, ensuring complete enumeration.

Combining (I) and (II) establishes the theorem. □

After establishing the soundness and completeness of Algorithm 2, we analyze the cost of the full normalization chain on which the algorithm depends. The growth arises from two stages: (i) *punctual expansion* in $PNF(\phi,k)$, which replaces each interval set by a Boolean combination of punctual literals up to k, and (ii) the *cross-product blow-up* caused by distributing conjunctions over the disjunctions introduced by *PNF*.

Theorem 3 (Complexity of full normalization)**.** *Let ϕ be a formula in μ*MTNL *and let $k \in \mathbb{N}$ be a finite time bound. The transformation $\phi \mapsto DNF(PNF(\phi,k))$ has worst-case space complexity exponential in both the average expansion factor of the interval sets in ϕ and the size of ϕ.*

Proof. We analyze the two stages separately.

Step 1: Punctual normalization. By Lemma 13, the size of $PNF(\phi,k)$ is bounded linearly in $|\phi|$ and in the average expansion factor $\iota_a(\phi)$:

$$|PNF(\phi,k)| \in \mathcal{O}\big(\iota_a(\phi)\cdot|\phi|\big).$$

Hence this step is polynomial in $|\phi|$ when $\iota_a(\phi)$ is bounded.

Step 2: Disjunctive normalization. Lemma 5 guarantees that every formula can be rewritten as an equivalent DNF $\phi' = \bigsqcup_i \phi'_i$, but distributing conjunctions over disjunctions introduces an exponential cross-product. Each obligation literal $O^{\mathfrak{I}}(\mathsf{a})$ in $PNF(\phi,k)$ yields a disjunction of $|\mathfrak{I}|$ punctual obligations, and each such disjunction multiplies the number of disjunct. Therefore, the number of product terms in $DNF(PNF(\phi,k))$ is exponential in the number of punctual obligations produced by *PNF*, which itself is linear in $\iota_a(\phi)\cdot|\phi|$.

Consequently,

$$|DNF(PNF(\phi,k))| \in \mathcal{O}\left(2^{\iota_a(\phi)\cdot|\phi|}\right),$$

and this bound characterizes the worst-case space of the normalization chain. □

3.4.3 Conflict Density

We now introduce a quantitative view of conflicts in μMTNL. The *conflict density* of a formula measures how frequently conflicts occur within its punctual representation. A high conflict density indicates a restrictive or overloaded specification: it reduces the set of feasible behaviors, complicates compliance checking, and complicates its interpretation. We obtain a simple diagnostic tool for refining normative specifications by the following formal definition.

Definition 30 (Conflict density)**.** *Let ϕ be a formula in μMTNL and let $k = \textsc{MaxOPoint}(\phi)$. The* conflict density *of ϕ, written $\mathsf{C}_\delta(\phi)$, is the ratio between the number of conflicting product terms and the total number of product terms in $DNF(PNF(\phi,k))$:*

$$\mathsf{C}_\delta(\phi) := \frac{|\mathcal{PC}(\phi)|}{\Xi(\phi)}.$$

Where:

- $|\mathcal{PC}(\phi)|$ *is the number of product terms identified as conflicting by Algorithm 2, and*
- $\Xi(\phi)$ *is the total number of product terms in $DNF(PNF(\phi,k))$.*

Conflict density captures the proportion of punctual obligations involved in conflicts. A formula with $\mathsf{C}_\delta(\phi) = 0$ is *conflict-free*; when $\mathsf{C}_\delta(\phi) = 1$ it is *fully conflicting*.

Example 37 (Conflict density computation)**.** *From Table Table 3.4 for Example 35, we have:*

$$\Xi(\phi) = 4, \qquad |\mathcal{PC}(\phi)| = 3.$$

Hence,

$$\mathsf{C}_\delta(\phi) = \frac{3}{4} = 0.75.$$

Thus, 75% of the product terms in ϕ contain conflicts, signaling an over-constrained specification.

Example 38 (Conflict density for UC)**.** *For our use case, Algorithm 1 reports 13 deontic punctual conflicts: five on the action* dap *at time points* $10, 11, 12, 13, 14$*, and eight on the action* dag *at each time point in* $[9, 16]$*. Consequently,*

$$|\mathcal{PC}(\mathsf{UC})| = 13, \qquad \Xi(\mathsf{UC}) = 16, \qquad \mathsf{C}_\delta(\mathsf{UC}) = \tfrac{13}{16} \approx 0.81.$$

About 81% *of the punctual obligations in* UC *are involved in conflicts.*

A formula with $\mathsf{C}_\delta(\phi) = 0$ is *conflict-free*; a formula with $\mathsf{C}_\delta(\phi) = 1$ is *fully conflicting* in the sense that every product term in its *DPNF* contains at least one punctual conflict. The following section lifts these punctual patterns and the conflict density measure to a proof-theoretic level, by relating conflicts in μMTNL to minimal unsatisfiable subsets.

3.5 Semantic Characterization and Elimination of Timed Normative Conflicts

This section lays the semantic basis for timed normative conflicts. We first recall propositional unsatisfiability, unsatisfiable cores, and minimal unsatisfiable subsets (MUS). We then transfer these notions to μMTNL by viewing product terms of the disjunctive punctual normal form (*DPNF*) as local clause sets. The key outcome is that punctual conflicts in μMTNL coincide with MUSes over punctual literals inside a single product term. This yields an exact assessment and explainability for formula unsatisfiability via conflict density and supports conflict-free rewriting. We close with the syntactic cost of conflict elimination.

3.5.1 Unsatisfiable Cores and MUS in Propositional Logic

A propositional formula F is *satisfiable* if some valuation makes F true, and *unsatisfiable* otherwise. If F is unsatisfiable, an *unsatisfiable core* is any subset $F' \subseteq F$ that is unsatisfiable. Cores isolate unsatisfiable sub-formulas but may contain clauses that are not necessary for proving unsatisfiability of the overall formula, as studied in [CGS07].

To address this limitation, the notion *minimal unsatisfiable subset* (MUS) refines unsatisfiable cores by requiring irreducibility: an MUS is an unsat core $U \subseteq F$ such that removing any clause makes U satisfiable. MUSes represent irreducible contradictions and are explored only when the studied formula is unsatisfiable. The structure and computation of MUSes have been widely studied [MS10, LS08], and it is well known that an unsatisfiable formula can contain several MUSes. A single clause may also appear in multiple MUSes, since different minimal combinations of clauses can independently give rise to unsatisfiability [LS05].

In our setting, this perspective is guided by the output of our conflict detection procedure (Algorithm 2): conflicts surface locally inside individual product terms of *DPNF*, not across the disjunction. We therefore analyze MUSes *per product term*, where the roles of clauses are played by punctual literals. The minimal unsatisfiable subsets of these literals are exactly the punctual conflicts that causes unsatisfiability in μMTNL.

3.5.2 From Propositional MUSes to Punctual MUSes in μMTNL

In the propositional setting, MUS analysis is performed on sets of clauses. A clause belongs to an MUS when it is part of the smallest subset of clauses that cannot all be true at the same time. In μMTNL, the surface syntax contains interval-based obligations and prohibitions, but after unfolding to disjunctive punctual normal form, the structure becomes closer to the propositional one. Let ϕ be a μMTNL formula. For $k = \textsc{MaxOPoint}(\phi)$,

$$DPNF(\phi, k) \;=\; \bigsqcup_i \mathsf{pt}_i$$

enumerates all product terms pt_i that are conjunctions of punctual obligations and prohibitions that may be active up to k. Each pt_i describes one complete punctual scenario: for each relevant time point and action it fixes whether the action is required, forbidden, or unconstrained. Satisfiability of pt_i is purely local: pt_i is satisfiable if and only if there exists a timed trace that meets all these punctual requirements.

This suggests the following correspondence. At the level of *DPNF*, the global formula ϕ behaves like a disjunction of local clause sets, one per product term. Unsatisfiability of ϕ means that *every* such local scenario is impossible, therefore every product term is unsatisfiable. Conversely, if some product term is satisfiable, then ϕ is satisfiable because

that term already provides a compliant behaviour. The natural analogue of a clause level MUS is then not a subset of the whole formula, but a minimal unsatisfiable subset of punctual literals *inside a single product term*. This is exactly what we call a punctual MUS.

Intuitively, a punctual MUS collects the smallest set of punctual prescriptions that cannot be jointly followed at the concrete times they refer to. From a normative point of view, it isolates the exact punctual situation where an agent faces an impossible choice. From a SAT viewpoint, it is the minimal unsatisfiable core at the level of punctual literals that sits inside one product term of the *DPNF*.

Definition 31 (Punctual MUS)**.** *Let $L = \{\ell_1, \ldots, \ell_n\}$ be punctual literals taken from a single product term* pt_i*. The conjunction $\bigsqcap_{\ell \in L} \ell$ is a* punctual minimal unsatisfiable subset *if:*

*(i) $\bigsqcap_{\ell \in L} \ell$ is unsatisfiable under the trace semantics of μ*MTNL*, and*

(ii) for every nonempty strict subset $L' \subset L$, the conjunction $\bigsqcap_{\ell \in L'} \ell$ is satisfiable.

Condition (i) states that the literals in L cannot all be enforced on any trace. Condition (ii) is the minimality requirement, it ensures that each literal in L is necessary for the unsatisfiability and that no smaller subset already conflicts. The definition is syntactic, since it speaks about inclusion between sets of literals, and semantic, since unsatisfiability is evaluated with respect to timed traces. Punctual MUSes therefore mirror classical MUSes, but are grounded at the level of punctual normative statements in μMTNL.

To understand the possible shape of punctual MUSes, we first look at the simplest building blocks. Single punctual literals never generate conflicts, they always have a trivial satisfying trace.

Lemma 19 (Satisfiability of isolated punctual literals)**.** *For any $a \in \Sigma$ and $t \in \mathbb{N}$, both $O^{\{[t,t]\}}(\mathsf{a})$ and $F^{\{[t,t]\}}(\mathsf{a})$ are satisfiable.*

Proof. Witnesses are $\langle (a,t) \rangle$ for $O^{\{[t,t]\}}(\mathsf{a})$ and $\langle - \rangle$ for $F^{\{[t,t]\}}(\mathsf{a})$. □

Conflicts, therefore, appear only when we combine punctual literals. We now show that there are exactly two kinds of minimal combinations that fail, which correspond to two very natural normative intuitions.

Lemma 20 (Deontic punctual conflict is a punctual MUS)**.** *For any $a \in \Sigma$ and $t \in \mathbb{N}$, the set $\{O^{\{[t,t]\}}(\mathsf{a}),\ F^{\{[t,t]\}}(\mathsf{a})\}$ is a punctual MUS.*

Proof. The conjunction is unsatisfiable, since no trace can both perform and avoid a at time t. Minimality follows from Lemma 19, each literal alone is satisfiable. □

Lemma 21 (Ontic punctual conflict is a punctual MUS)**.** *For any distinct $a, b \in \Sigma$ and any $t \in \mathbb{N}$, the set $\{O^{\{[t,t]\}}(\mathsf{a}),\ O^{\{[t,t]\}}(\mathsf{b})\}$ is a punctual MUS.*

Proof. The conjunction is unsatisfiable, since the trace semantics admits at most one action at time t. Again, by Lemma 19 each singleton is satisfiable, so the pair is minimal. □

These two lemmas show that direct normative conflicts at a point in time (obligate and forbid the same action) and physical or behavioural impossibility at a point in time (obligate two distinct actions that cannot both happen) are punctual MUSes. The next lemma shows that there are no other punctual MUS patterns.

Lemma 22 (Completeness of punctual MUS types)**.** *Every punctual minimally unsatisfiable subset in μMTNL is either a deontic or an ontic punctual conflict.*

Proof. Exhaust over the identity of the actions and the equality of the time points. If the time points differ, the literals constrain different instants and can always be satisfied together. If they concern the same action at the same time, pairs of the form $O^{\{[t,t]\}}(\mathsf{a})$ with $O^{\{[t,t]\}}(\mathsf{a})$ and $F^{\{[t,t]\}}(\mathsf{a})$ with $F^{\{[t,t]\}}(\mathsf{a})$ are redundant but satisfiable, while the mixed pair $O^{\{[t,t]\}}(\mathsf{a})$ with $F^{\{[t,t]\}}(\mathsf{a})$ produces a deontic punctual conflict.

If they concern distinct actions at the same time, the pair $O^{\{[t,t]\}}(\mathsf{a})$ with $O^{\{[t,t]\}}(\mathsf{b})$ yields an ontic punctual conflict by the single–action–per–time semantics, whereas any combination that includes at least one prohibition, such as $O^{\{[t,t]\}}(\mathsf{a})$ with $F^{\{[t,t]\}}(\mathsf{b})$, remains satisfiable, since prohibitions do not force any action to occur.

□

Soundness follows from Lemmas 20 and 21, and completeness from Lemma 22. Every punctual MUS in μMTNL is therefore either deontic or ontic. Algorithmically, this means that conflict detection at the punctual level reduces to a simple pattern matching task inside product terms of *DPNF*. Conceptually, it shows that all punctual minimal unsatisfiable subsets in μMTNL can be read as either "ought and ought not" for the same action at a time, or "ought and ought" for two incompatible actions at a time.

3.5.3 Unsatisfiability in Terms of Conflict Density

Let $\mathsf{C}_\delta(\phi)$ denote the fraction of product terms in $\mathit{DPNF}(\phi,k)$ that contain a punctual conflict. Write $\Xi(\phi)$ for the number of product terms and $\mathcal{PC}(\phi)$ for the set of conflicting product terms, so $\mathsf{C}_\delta(\phi) = |\mathcal{PC}(\phi)|/\Xi(\phi)$.

Lemma 23 (Fully conflicting implies unsatisfiable)**.** *If* $\mathsf{C}_\delta(\phi) = 1$*, then* ϕ *is unsatisfiable.*

Proof. Every product term in $\mathit{DPNF}(\phi,k)$ is unsatisfiable only if it contains a conflict. When the formula is fully conflicting it means that the number of conflicting product terms equals the number of product terms returned by the $\mathit{DPNF}(\phi,k)$, as each one is unstisfiable then the conjunction of n unsatisfiable product terms is also unsatisfiable as showed in Lemma 10. □

Theorem 4 (Punctual MUS structure of unsatisfiable product terms)**.** *Let* pt *be a product term in* $\mathit{DPNF}(\phi,k)$*. Then* pt *is unsatisfiable iff it contains a punctual deontic or ontic conflict. Moreover, each such conflict forms a minimal unsatisfiable subset of punctual literals.*

Proof sketch. If pt is unsatisfiable, minimality of the unsatisfiable conjunction of literals implies the existence of an unsatisfiable pair of punctual literals. By Lemma 22, the only such pairs are the deontic or ontic patterns. Conversely, any deontic or ontic punctual conflict yields an unsatisfiable and inclusion-minimal product term, since removing literals in MUS restores satisfiability. □

Corollary 2 (Unsatisfiable implies full conflict)**.** *If* ϕ *is unsatisfiable, then* $\mathsf{C}_\delta(\phi) = 1$*.*

Proof. ϕ is the disjunction of its product terms. If ϕ is unsatisfiable, each product term is unsatisfiable. By Theorem 4, each contains a punctual conflict, so every term is counted in $\mathcal{PC}(\phi)$. □

Theorem 5 (Exact characterization of unsatisfiability)**.** *A formula* $\phi \in \mu\mathsf{MTNL}$ *is unsatisfiable if and only if* $\mathsf{C}_\delta(\phi) = 1$*.*

Proof. Combine Lemma 23 with Corollary 2. □

3.5.4 Semantic Preserving Conflict-free Pruning and Induced Verbosity

When $\mathsf{C}_\delta(\phi) < 1$, at least one product term is conflict-free. Pruning all conflicting terms yields an equivalent conflict-free disjunction.

Theorem 6 (Conflict-free rewriting)**.** *If* $\mathsf{C}_\delta(\phi) \neq 1$*, there exists* ϕ' *with* $\phi \equiv \phi'$ *and* $\mathsf{C}_\delta(\phi') = 0$.

Proof sketch. Compute $k = \textsc{MaxOPoint}(\phi)$ and $\phi_{\text{dpnf}} = \bigsqcup_i \mathsf{pt}_i$. Use Algorithm 2 to classify product terms. Let $I = \{i \mid \mathsf{pt}_i \text{ has no punctual conflict}\}$. Set $\phi' = \bigsqcup_{i \in I} \mathsf{pt}_i$. Removed terms are unsatisfiable, so $\phi \equiv \phi'$, and by construction $\mathsf{C}_\delta(\phi') = 0$. □

Algorithm 3: Conflict-Free Rewriting *CFR*

Input: $\phi \in \mu\mathsf{MTNL}$
Output: **False** if all terms conflict, else a conflict-free $\mathit{CFR}(\phi)$

$k \leftarrow \textsc{MaxOPoint}(\phi)$; $\quad \phi_{\text{dpnf}} \leftarrow \mathit{DNF}(\mathit{PNF}(\phi, k))$; $\quad \mathcal{CF} \leftarrow \emptyset$;
foreach *product term* pt *in* ϕ_{dpnf} **do**
  if pt *contains no deontic or ontic punctual conflict* **then**
    $\mathcal{CF} \leftarrow \mathcal{CF} \cup \{\mathsf{pt}\}$
if $\mathcal{CF} = \emptyset$ **then**
  return False
$\mathit{CFR}(\phi) \leftarrow \bigsqcup_{\mathsf{pt} \in \mathcal{CF}} \mathsf{pt}$; **return** $\mathit{CFR}(\phi)$

Conflict elimination can enlarge the formula. In particular, avoiding simultaneous obligations at the same time forces *DNF* to retain only assignments with pairwise distinct time points, which inflates the disjunction.

Example 39.

$$\phi := O^{\{[0,3]\}}(\mathsf{a}) \sqcap O^{\{[0,3]\}}(\mathsf{b}) \sqcap O^{\{[0,3]\}}(\mathsf{c}).$$

Table Table 3.5 summarizes the transformation pipeline—PNF, DNF, conflict extraction, and the conflict-free rewriting $\mathit{CFR}(\phi)$*. The final formula is roughly* $20\times$ *larger than the original.*

Intuitively, $\mathit{DNF}(\mathit{PNF}(\phi, 3))$ *enumerates all* 4^3 *assignments of times to* $(\mathsf{a}, \mathsf{b}, \mathsf{c})$*. To avoid conflicts, only combinations with pairwise distinct times remain, i.e.* $4 \cdot 3 \cdot 2 = 24$ *product terms.*

Step	Expression	Size
ϕ	$O^{\{[0,3]\}}(\mathsf{a}) \sqcap O^{\{[0,3]\}}(\mathsf{b}) \sqcap O^{\{[0,3]\}}(\mathsf{c})$	18
$PNF(\phi,3)$	$\left(\bigsqcup_{i=0}^{3} O^{\{[i,i]\}}(\mathsf{a})\right) \sqcap \left(\bigsqcup_{i=0}^{3} O^{\{[i,i]\}}(\mathsf{b})\right) \sqcap \left(\bigsqcup_{i=0}^{3} O^{\{[i,i]\}}(\mathsf{c})\right)$	59
$DNF(PNF(\phi,3))$	$\bigsqcup_{0\le i,j,k\le 3} \left(O^{\{[i,i]\}}(\mathsf{a}) \sqcap O^{\{[j,j]\}}(\mathsf{b}) \sqcap O^{\{[k,k]\}}(\mathsf{c})\right)$	959
Punctual conflicts	$\left\{O^{\{[t,t]\}}(\mathsf{a}) \sqcap O^{\{[t,t]\}}(\mathsf{b}), O^{\{[t,t]\}}(\mathsf{a}) \sqcap O^{\{[t,t]\}}(\mathsf{c}), O^{\{[t,t]\}}(\mathsf{b}) \sqcap O^{\{[t,t]\}}(\mathsf{c}) \mid t \in \{0,1,2,3\}\right\}$	–
$CFR(\phi)$	$\bigsqcup_{\substack{i,j,k\in\{0,1,2,3\} \\ \text{pairwise distinct}}} \left(O^{\{[i,i]\}}(\mathsf{a}) \sqcap O^{\{[j,j]\}}(\mathsf{b}) \sqcap O^{\{[k,k]\}}(\mathsf{c})\right)$	359
$\lvert CFR(\phi)\rvert / \lvert\phi\rvert$	Blow-up factor	≈ 20

Table 3.5: Conflict-elimination size comparison for $\phi := O^{\{[0,3]\}}(\mathsf{a}) \sqcap O^{\{[0,3]\}}(\mathsf{b}) \sqcap O^{\{[0,3]\}}(\mathsf{c})$.

Although this example illustrates worst case blow-ups, conflict-free formulas often admit concise equivalents by *merging* punctual literals over the same action. Disjunctions of punctual obligations can be compacted via **ObDis**, and conjunctions of punctual prohibitions via **FConj**.

Example 40.

$$\mathsf{UC}_2' := O^{\{[9,16]\}}(\mathsf{dap}) \sqcap F^{\{[10,14]\}}(\mathsf{dap}) \sqcap F^{\{[2,4],[8,24]\}}(\mathsf{dag}).$$

The conflict-free rewriting $CFR(\mathsf{UC}_2')$ *retains the non-conflicting obligations* $t \in \{9,15,16\}$. *Table Table 3.6 compares the original, verbose rewriting, and a merged equivalent* UC_2''.

Formula	Expression	Size
UC_2'	$O^{\{[9,16]\}}(\mathsf{dap}) \sqcap F^{\{[10,14]\}}(\mathsf{dap}) \sqcap F^{\{[2,4],[8,24]\}}(\mathsf{dag})$	16
$CFR(\mathsf{UC}_2')$	$\bigsqcup_{t\in\{9,15,16\}} \Big(O^{\{[t,t]\}}(\mathsf{dap}) \sqcap \bigsqcap_{u\in[10,14]} F^{\{[u,u]\}}(\mathsf{dap}) \sqcap \bigsqcap_{v \subseteq \{[2,4],[8,16]\}} F^{\{[v,v]\}}(\mathsf{a}) \sqcap F^{\{[17,24]\}}(\mathsf{a})\Big)$	94
UC_2''	$O^{\{[9,9],[15,16]\}}(\mathsf{dap}) \sqcap F^{\{[10,14]\}}(\mathsf{dap}) \sqcap F^{\{[2,4],[8,24]\}}(\mathsf{dag})$	18

Table 3.6: Refactoring a verbose conflict-free rewriting $CFR(\mathsf{UC}_2')$ into a compact equivalent UC_2''.

Here UC_2'' *merges the surviving punctual obligations on* dap *into a single interval set, while prohibitions on* dag *are recombined into their original form.*

These examples motivate a strategy of *controlled conflict-density management*: When conciseness is essential, designers may either (i) keep compact high-level formulas that implicitly contain ontic conflicts and defer their resolution to planning, or (ii) eliminate conflicts up front and refactor the resulting *DPNF* with merging rules (**ObDis**, **FConj**) to restore compactness. In the next section, we introduce a generalized punctual conflict rule that unifies deontic and ontic cases and enables concise and big-step reasoning without requiring any *DPNF* rewriting.

3.6 Conflict Calculus over Generalized Rules

The previous section showed that eliminating punctual conflicts can cause substantial growth in formula size. While some verbose conflict-free rewriting can be refactored compactly, others cannot without losing semantic precision. We now generalize the conflict patterns observed earlier and formalize them as inference rules. These rules form a *conflict calculus*, enabling systematic exploration of formulas and selective conflict removal.

Table Table 3.7 summarizes the syntactic notations of our deductive system, grounded in the extensional semantics of μMTNL. We use standard derivability and equivalence symbols, with their semantic counterparts defined over duty classes.

Notation	Meaning	Semantic mapping
$\phi \vdash \bot$	ϕ is unsatisfiable	$[\![\phi]\!] = \emptyset$
$\vdash \phi$	ϕ is valid	$[\![\phi]\!] = 2^{\mathbb{T}}$
$\phi \vdash \phi'$	ϕ entails ϕ'	$[\![\phi]\!] \subseteq [\![\phi']\!]$
$\phi \equiv \phi'$	ϕ is equivalent to ϕ'	$[\![\phi]\!] = [\![\phi']\!]$

Table 3.7: Syntactic notations and their semantic mapping.

3.6.1 Generalized Conflicts Management

We now lift the punctual conflict analysis to a higher level of abstraction. Rather than changing our fine-grained definition of conflicts, the following lemmas provide *generalized management rules* that operate on non-atomic literals. They consolidate multiple punctual conflicts into larger reasoning steps, allowing one to handle clusters of conflicting intervals in a single inference. These big-step rules cover two fundamental forms of

normative tension: *deontic* and *ontic* conflicts. Each lemma specifies the structural conditions under which such compound conflicts arise and how they can be either resolved or rewritten equivalently.

Lemma 24 (Generalized management of Deontic Conflicts)**.** *Let $\mathfrak{I}$ and $\mathfrak{I}'$ be interval sets and* a *an action.*

(i) If $\mathfrak{I} \subseteq \mathfrak{I}'$, then:

$$O^{\mathfrak{I}}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a}) \vdash \bot.$$

(ii) If $\mathfrak{I} \cap \mathfrak{I}' \neq \emptyset$ and $\mathfrak{I} \not\subseteq \mathfrak{I}'$, then:

$$O^{\mathfrak{I}}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a}) \equiv O^{\mathfrak{I} \ominus \mathfrak{I}'}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a}).$$

Case (i) expresses a *total deontic conflict*: an obligation and a prohibition on the same action and time frame cannot coexist, making the conjunction unsatisfiable. Case (ii) captures a *partial deontic conflict*, where the overlap between obligation and prohibition is only partial; the formula remains satisfiable after removing the conflicting portion.

Proof. **(i) T-DeoConf.** If $\mathfrak{I} \subseteq \mathfrak{I}'$, each required point $t \in \mathfrak{I}$ is also forbidden at $t \in \mathfrak{I}'$, yielding punctual conflicts $O^{\{[t,t]\}}(\mathsf{a}) \sqcap F^{\{[t,t]\}}(\mathsf{a})$. No trace can satisfy both, hence the conjunction is unsatisfiable.

(ii) P-DeoConf. When $\mathfrak{I} \cap \mathfrak{I}' \neq \emptyset$ and $\mathfrak{I} \not\subseteq \mathfrak{I}'$, only points in $\mathfrak{I} \cap \mathfrak{I}'$ are conflicting. Obligations outside the overlap remain valid. Removing the conflicting portion yields the equivalent formula:

$$O^{\mathfrak{I}}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a}) \equiv O^{\mathfrak{I} \ominus \mathfrak{I}'}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a}).$$

□

Lemma 25 (Generalized management of Ontic Conflict)**.** *Let $\mathfrak{I}$ be a finite interval set and $\{a_1, \ldots, a_n\}$ a set of distinct actions such that $|\mathfrak{I}| <_\infty |\{a_1, \ldots, a_n\}|$. Then:*

$$\prod_{i=1}^{n} O^{\mathfrak{I}}(\mathsf{a_i}) \vdash \bot.$$

This rule captures the situation where more actions are obliged than available time points make simultaneous fulfilment impossible. Partial ontic rewritings are intentionally omitted, as discussed earlier, due to combinatorial explosion.

Proof. Let $|\mathfrak{I}| = m < n$. Each $O^{\mathfrak{I}}(\mathsf{a_i})$ expands as $\bigsqcup_{t\in\mathfrak{I}} O^{\{[t,t]\}}(\mathsf{a_i})$; their conjunction enumerates all assignments of actions to time points. Since $n > m$, some t must host at least two actions a_i, a_j, producing punctual ontic conflicts $O^{\{[t,t]\}}(\mathsf{a_i}) \sqcap O^{\{[t,t]\}}(\mathsf{a_j}) \vdash \bot$. By the pigeonhole principle [Dir63, Ajt94], every disjunct is unsatisfiable, hence the whole formula is unsatisfiable. □

3.6.2 Conflict Calculus

The conflict calculus in Figure 3.4 consolidates nine inference rules previously derived as lemmas. Together they form a structured system for detecting, rewriting, and resolving normative conflicts.

- **(O-Union)** and **(F-Union)** merge adjacent or disjoint obligations and prohibitions into single literals, simplifying formulas after resolution.
- **(⊥⊓)** and **(⊥⊔)** propagate unsatisfiability through conjunctions and disjunctions.
- **(⊔⊥)** prunes unsatisfiable branches in disjunctions.
- **(⊥ ≡)** and **(≡ ⊓)** preserve (un)satisfiability under equivalence substitution.
- **(Distr1L)** distributes conjunction over disjunction to produce disjunctive normal form.
- **(T-DeoConf)** and **(P-DeoConf)** identify total and partial deontic conflicts, the latter enabling safe rewriting of overlapping intervals.
- **(OntConf)** generalizes punctual ontic conflicts, deriving unsatifiability when obligations outnumber time points.

Example 41 (Analyzing UC via Conflict Calculus)**.** *Let*

$\mathsf{UC}_1' := O^{\{[9,16]\}}(\mathsf{dag}) \sqcap \mathsf{MA} \sqcap \mathsf{SR}, \mathsf{UC}_2' := O^{\{[9,16]\}}(\mathsf{dap}) \sqcap \mathsf{MA} \sqcap \mathsf{SR}$
and $\mathsf{UC} := \mathsf{UC}_1' \sqcup \mathsf{UC}_2'$ *with* $\mathsf{MA} := F^{\{[2,4],[8,24]\}}(\mathsf{dag}), \quad \mathsf{SR} := F^{\{[10,14]\}}(\mathsf{dap})$.
A constructive way of prunning UC *with be following the next three steps:*
Step 1: Total deontic conflict in UC_1'**.**

$$\dfrac{\dfrac{}{O^{\{[9,16]\}}(\mathsf{dag}) \sqcap F^{\{[2,4],[8,24]\}}(\mathsf{dag}) \vdash \bot}\,(\textit{T-DeoConf})}{\mathsf{UC}_1' \vdash \bot}\,(\bot\sqcap)$$

Thus UC_1' *is unsatisfiable due to overlapping obligation and prohibition intervals on the same action.*

Step 2: Partial deontic conflict in UC_2'**.**

$$\textbf{(O-Union)}\quad \frac{\mathfrak{I} = \mathfrak{I}_1 \cup \mathfrak{I}_2}{O^{\mathfrak{I}_1}(\mathsf{a}) \sqcup O^{\mathfrak{I}_2}(\mathsf{a}) \equiv O^{\mathfrak{I}}(\mathsf{a})}$$

$$\textbf{(F-Union)}\quad \frac{\mathfrak{I} = \mathfrak{I}_1 \cup \mathfrak{I}_2}{F^{\mathfrak{I}_1}(\mathsf{a}) \sqcup F^{\mathfrak{I}_2}(\mathsf{a}) \equiv F^{\mathfrak{I}}(\mathsf{a})}$$

$$\textbf{(Distr1L)}\quad \frac{}{\phi_1 \sqcap (\phi_2 \sqcup \phi_3) \equiv (\phi_1 \sqcap \phi_2) \sqcup (\phi_1 \sqcap \phi_3)}$$

$$\textbf{(}\equiv\sqcap\textbf{)}\quad \frac{\phi_1 \equiv \phi_1'}{\phi_1 \sqcap \phi_2 \equiv \phi_1' \sqcap \phi_2} \qquad \textbf{(}\bot\equiv\textbf{)}\quad \frac{\phi' \vdash \bot \quad \phi \equiv \phi'}{\phi \vdash \bot}$$

$$\textbf{(}\bot\sqcap\textbf{)}\quad \frac{\phi \vdash \bot}{\phi \sqcap \phi' \vdash \bot}$$

$$\textbf{(}\bot\sqcup\textbf{)}\quad \frac{\phi \vdash \bot \quad \phi' \vdash \bot}{\phi \sqcup \phi' \vdash \bot} \qquad \textbf{(}\sqcup\bot\textbf{)}\quad \frac{\phi' \vdash \bot}{\phi \sqcup \phi' \equiv \phi}$$

$$\textbf{(T-DeoConf)}\quad \frac{\mathfrak{I} \subseteq \mathfrak{I}'}{O^{\mathfrak{I}}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a}) \vdash \bot}$$

$$\textbf{(OntConf)}\quad \frac{|\mathfrak{I}| <_\infty |\{a_1, \ldots, a_n\}|}{\bigsqcap_{a_i \in \{a_1,\ldots,a_n\}} O^{\mathfrak{I}}(\mathsf{a_i}) \vdash \bot}$$

$$\textbf{(P-DeoConf)}\quad \frac{\mathfrak{I} \cap \mathfrak{I}' \neq \emptyset \quad \mathfrak{I} \not\subset \mathfrak{I}'}{O^{\mathfrak{I}}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a}) \equiv O^{\mathfrak{I} \ominus \mathfrak{I}'}(\mathsf{a}) \sqcap F^{\mathfrak{I}'}(\mathsf{a})}$$

Figure 3.4: Conflict calculus for μMTNL

$$\dfrac{\dfrac{}{O^{\{[9,16]\}}(\mathsf{dap}) \sqcap F^{\{[10,14]\}}(\mathsf{dap}) \equiv O^{\{[9,9],[15,16]\}}(\mathsf{dap}) \sqcap F^{\{[10,14]\}}(\mathsf{dap})}\ \textit{(P-DeoConf)}}{\mathsf{UC}_2' \equiv \mathsf{UC}_2}\ \textit{(}\equiv\sqcap\textit{)}$$

UC_2' *is thus satisfiable after eliminating overlapping intervals.*

Step 3: Combined derivation for UC**.**

$$\frac{\mathsf{UC}_1' \vdash \bot \qquad \mathsf{UC}_2' \equiv \mathsf{UC}_2}{\mathsf{UC}_1' \sqcup \mathsf{UC}_2' \equiv \mathsf{UC}_2}\ \textit{(}\bot\sqcup\textit{)}$$

Hence the global formula UC *is equivalent to its conflict free refinement* UC_2.

The conflict calculus thus provides a uniform reasoning layer over μMTNL: each rule isolates minimal sources of unsatisfiability, supports controlled rewriting, and preserves

equivalence where possible. Its proof-theoretic structure also makes it directly implementable in theorem provers such as Coq, Isabelle, or Lean, where each rule can be encoded as a tactic enabling guided, reproducible proof search. This opens a path toward an interactive assistant for designing consistent normative systems, capable of explaining, verifying, and refining specifications through structured derivations.

3.7 Discussion of the Overall Framework

This section situates our technical contributions within a structured methodology for analyzing and resolving conflicts in timed normative specifications. We first highlight the formal pipeline of results, which connects definitions, lemmas, algorithms, and theorems into a coherent workflow. We then explain how the framework can be used in practice, showing how each component interacts to refine normative specifications and support human-in-the-loop reasoning.

3.7.1 Answering the Research Questions Through Formal Results

This section situates our technical contributions within a structured methodology for analyzing and resolving conflicts. We demonstrate how the formal pipeline—connecting definitions, algorithms, and theorems—provides precise answers to the three research questions (RQ1–RQ3) posed in Section Section 3.1.

Answering RQ1: The Relation between Unsatisfiability and Conflict

RQ1 asked: *Is every unsatisfiable formula a conflict? Conversely, does the presence of a conflict necessarily imply that the formula is unsatisfiable?*

Our framework establishes a bidirectional equivalence between semantic unsatisfiability and the presence of syntactic conflict patterns, provided the analysis is performed at the level of the Disjunctive Punctual Normal Form (*DPNF*).

- **From Conflict to Unsatisfiability:** We first defined *Punctual Conflicts* (Definition 24) as atomic contradictions occurring at specific time points. We proved that any product term containing such a conflict is necessarily unsatisfiable (Theorem 4). Furthermore, if a formula is *fully conflicting* (i.e., every disjunct in its normal form contains a conflict), it is unsatisfiable (Lemma 23).

- **From Unsatisfiability to Conflict:** Conversely, we proved that every unsatisfiable product term must contain a deontic or ontic punctual conflict as a Minimal Unsatisfiable Subset (MUS) (Theorem 4).
- **The Exact Characterization:** We consolidated these results into **Theorem 5**, which provides the exact characterization: A formula ϕ is unsatisfiable if and only if its *Conflict Density* is exactly 1.

Answer: Yes. In μMTNL, unsatisfiability is strictly equivalent to being fully conflicting. Every logical inconsistency can be traced back to atomic deontic or ontic conflict patterns.

Answering RQ2: Conflict Quantification

RQ2 asked: *Can we define a quantitative measure for the degree of conflict within a specification?*

We moved beyond binary satisfiability checks by formalizing a continuous measure of inconsistency.

- **Metric Definition:** We introduced *Conflict Density* ($\mathsf{C}_\delta(\phi)$) in **Definition 30**. This metric calculates the ratio between the number of conflicting product terms found by **Algorithm 2** and the total number of scenarios in the specification's normal form.
- **Granularity:** This measure relies on the systematic enumeration of conflicts enabled by the *Disjunctive Punctual Normal Form* (*DPNF*) (Definition 22) and the *Punctual Normal Form* (Definition 29), which decompose complex constraints into countable atomic units.

Answer: Yes. We quantify the degree of conflict as a rational number $\mathsf{C}_\delta(\phi) \in [0,1]$, representing the proportion of infeasible execution scenarios in the specification.

Answering RQ3: Properties-Preserving Resolution

RQ3 asked: *Is it always possible to resolve conflicts in a formula without altering its inherent semantics or compromising its structural compactness?*

Our framework provides a mixed answer: semantic preservation is guaranteed, but structural compactness is subject to trade-offs.

- **Semantic Preservation:** We proved that for any formula with $\mathsf{C}_\delta(\phi) < 1$, there exists a *Conflict-Free Rewriting* (*CFR*(ϕ)) that is semantically equivalent to the satisfiable subset of the original formula (**Theorem 6**). This is implemented via **Algorithm 3**, which prunes exactly those branches identified as conflicting.

- **Structural Compactness:** We showed that this resolution is *not* always compact. **Theorem 3** establishes that the full normalization required for conflict elimination can induce an exponential blow-up in formula size.
- **Mitigation:** However, we demonstrated that compactness can often be restored using *Literal Compression rules* (Lemma 6) and the *Conflict Calculus* (Figure 3.4). These tools allow for big-step manual simplifications (e.g., Lemma 24) to mitigate the verbosity of the automated output. We also suggested the conflict calculus that enables one to choose which conflicts to use to avoid size exlosion. The conflict calculus has no rule for eliminating ontic conflicts as it has an important blow-up factor.

Answer: We can always resolve conflicts while preserving the intended *compliant behaviors* (semantics). However, we cannot guarantee *structural compactness*; conflict elimination may increase the size of the specification, necessitating a post-processing refactoring step.

3.7.2 How to Use the Framework Reasoning Tools

The possible usages of the framework's tools is illustrated in Figure 3.5. It begins with a normative specification ϕ, which undergoes both logical and feasibility analysis. A key starting point is the SMT-based time infeasibility check, which identifies whether the timing constraints in the specification are jointly satisfiable. The specification could also be translated into Disjunctive Punctual Normal Form (DPNF). This transformation enables fine-grained conflict detection and explanation by reducing complex temporal constructs into atomic, punctual obligations and prohibitions.

Conflict detection then classifies inconsistencies into ontic and deontic types. The structural transformations inform this process applied through DPNF and return the set of punctual conflicts $\mathcal{PC}(\phi)$ as well as the conflict density measure $\mathsf{C}_\delta(\phi)$. Once identified, conflicts are passed to the conflict elimination phase, which removes all product terms containing conflicts. As discussed previously, this automatic conflict elimination procedure can result in returning a refined specification ϕ' that does not contain any conflict but may have a bad conciseness ratio.

The conflict calculus provides the user with the option for a tailored trade-off and simpler reasoning while preserving soundness to the conflict detection algorithm. It offers a higher-level analysis of normative specification without going to the disjunctive punctual normal form, which is too verbose for human processing. It enables formal reasoning through rule-based derivations, allowing designers to manually prune, rewrite, or simplify problematic portions of a specification to reduce the conflict density measure while maintaining a desired concise ratio compared to the original normative specification. The

calculus is also bidirectionally connected to the refined normative specification ϕ', as it both informs its construction and enables subsequent updates or revisions. This interactive loop reflects real-world design cycles where specifications evolve iteratively under refinement, correction, or contextual adaptation. Unless all the other tools in the framework are. The calculus is not automated and requires the user to understand the semantics and rules of the proof system but offers him shorter explanation and big steps reasoning.

The conflict density measure component receives information from the conflict detection phase to quantify the degree of normative uncertainty inherent in the original specification. This metric supports informed decision-making by offering a way to compare alternative encodings or prioritize simplification steps.

Ultimately, the framework yields a refined normative specification ϕ' that may be conflict-free and, ideally, more amenable to implementation or automated reasoning. The conciseness ratio $\frac{|\phi'|}{|\phi|}$ evaluates the trade-off between conflict elimination and formula size. In scenarios where full normalization introduces syntactic blow-up, this ratio provides helpful guidance for designers balancing readability and conflict elimination.

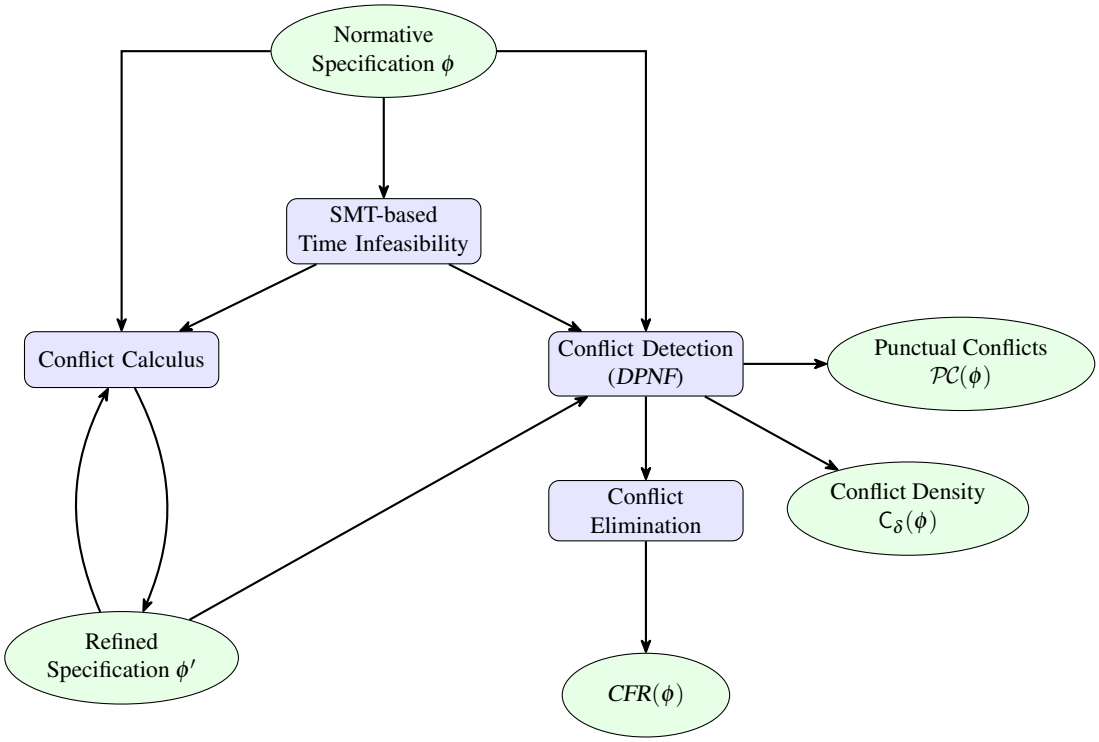


Figure 3.5: Diagram summarizing the conflict analysis and resolution framework.

3.8 Related Work

To our knowledge, this is the first framework to integrate systematically: (i) a metric-time deontic logic with action-based semantics; (ii) structural use of interval sets for simplifying formula representations; (iii) algorithmic techniques for detecting and eliminating conflicts; (iv) a conflict calculus that supports formal, traceable derivations; and (v) a quantitative analysis of conflict density and conciseness in normative specifications and their refinements.

Existing works suggests many ways to define and detect normative conflicts with temporal reasoning. The majority of current frameworks either limit themselves to discrete-time models or employ only axiomatic approaches without providing operational steps for handling conflict. Our framework differs, introducing a syntactically sound and semantically founded methodology that incorporates normalization, algorithmic detection, conflict elimination, and formal conflict calculus. The following explanations make clear how our research compares to previous work.

Conflicts Characterization Pace and Schapachnik [Pac20] propose logical criteria for identifying conflicts in contracts using temporal deontic logic. Their framework operates in a discrete-time setting, where obligations are defined over abstract steps rather than metric time points. Notably, their primary focus lies in environmental conflicts, which is another type of ontic conflicts related to the environement constraint rather than the agent. A typical example involves a contract that obliges an agent to perform actions a, b, and c, while an external constraint prohibits executing all three simultaneously. These conflicts are not necessarily due to temporal contradictions, as we consider in our work when multiple obligations are assigned to distinct actions at the same time point, but rather arise from feasibility limitations imposed by the environment.

A closer comparison can be drawn with the work of Colombo and Governatori [CTGK14], who present a temporal semantic framework for analyzing various types of obligations, standard, achievement, maintenance, preemptive, and compensable—using timed traces as semantic models. Their framework captures a broader typology of conflicts, including what we define as deontic conflicts as well as achievement/achievement conflicts, which in our setting correspond to ontic conflicts. While their use of timed traces enables reasoning about the evolution of obligations over time, timing constraints are not directly embedded in the logical syntax. Instead, they are modeled via external annotations such as triggers and deadlines. This externalization of timing has significant implications. For example, in their setting, conflict detection relies on manually specifying incompatibilities between actions such as declaring that “going to the bar” and “writing a paper” cannot occur simultaneously. This is necessary because actions are represented as atomic propositions, all of which may co-occur at the same time point unless explicitly constrained. In

contrast, our logic internalizes timing constraints as first-class constructs. Conflicts arise not from assumed action incompatibilities but from the interaction of metric intervals within which obligations and prohibitions are specified. This intrinsic modeling of time allows us to detect inconsistencies purely based on the temporal semantics of the norms, without requiring auxiliary assumptions about the actions themselves. We acknowledge that, in practice, some actions can be concurrent; our minimal ontic assumption can be relaxed by replacing the single–action capacity with a general feasibility predicate. Thus, an expressive ontic layer lies between the two settings: Under this regime, exceptions are positive, one can explicitly declare which actions are co-executable, rather than attempting to enumerate all non-compatible pairs.

A framework is presented in [FMGY$^+$24] that models temporal constraints using deadlines, i.e, single time point within which a prescribed event must or must not occur (e.g., "open the curtain within 30 minutes"). These constraints are not expressed as extended intervals such as $[10, 14]$ or sets of disjoint intervals, but rather as single continuous windows anchored to specific triggering conditions. The framework does not reason over arbitrary collections of time periods; instead, it focuses on whether an event satisfies its associated deadline constraint. Within this setting, they distinguish two forms of normative conflict: vacuous conflicts, which arise when a rule can never be applied without violating another, for instance, one rule obliges opening the curtains while another forbids it in all cases; and situational conflicts, which emerge only under certain conditions, where otherwise non-conflicting rules issue contradictory demands when triggered simultaneously in a specific context.

Conflict Elimination A comparable strategy for conflict elimination in timed normative specifications is found in the work [TMRG21], which introduces a belief revision framework for legal systems based on temporalised logic. Their approach also addresses normative systems with discrete time, allowing for fine-grained control over the temporal applicability of norms. At the core of their framework lies a temporal revision operator, denoted σ, which maintains consistency when new normative clause is introduced. Rather than removing the entire conflicting rules, the operator identifies the specific temporal fragments,i.e., sub-intervals, responsible for the contradiction and selectively trims them.

In contrast, our approach differs in several key respects: (i) it addresses not only deontic conflicts but also ontic conflicts, as our logic is action-based; (ii) our logic does not support conditionals, while they do not consider disjunction; (iii) our conflict detection and elimination revises the entire normative system as a whole, rather than prioritizing the newly introduced norm over the existing ones; (iv) it leverages interval sets to yield a more concise and refined normative specification, whilst they solely rely on intervals; and (v) our approach is semantic and syntax-based, equipped with algorithms and conflict calculus.

3.9 Conclusion

We presented a micro metric-time normative logic, μMTNL, together with a pipeline for *explainable* conflict analysis: syntactic normalization to disjunctive/punctual forms, a sound QF-LIA encoding for timed feasibility, a constructive conflict-extraction procedure, and a proof-theoretic conflict calculus. By internalising metric time via canonical interval sets and making the ontic resource constraint explicit (*one action per time point*), conflicts arise transparently from the interaction of time-bounded norms and that constraint. Beyond detecting conflict, we contributed additionally by: (i) defining a *conflict-density* measure and proving a *pruning theorem* that rewrites any partially conflicting specification into a semantically equivalent, conflict-free refinement; and (ii) complexity/algorithms for μMTNL satisfiability is NP-complete with a tight QF-LIA encoding bounded by the number of obligations, together with explicit size blow-up bounds for punctualisation (showing the conciseness vs. conflict-elimination trade-off).

Transferability. We believe that our methodology is transferable to many other existing logics: more precisely agent-centred timed deontic formalisms and other contract specification languages beyond μMTNL, providing a systematic approach to conflict detection and verifiable repair in contracts, autonomous agents, and legal compliance.

Outlook. We plan two enrich μMTNL gradually and in two different directions: (1) *expressivity*: actions with duration and ought to be literals (defined on states), reparation operator, conditional/permission literals and a defeasibility layer; and (2) *interaction*: frames for prevention/enabling/interference between actions, and relaxing the single-action ontic constraint. We expect the second direction to require substantial methodological changes for the full logic specified in [KLS21], since it requires additional structural properties on the underlying frames used to model the ontical properties of the enriched logic.

4 Reasoning about Blame in Multi-Party Collaborative Contracts

This chapter addresses the challenge of monitoring and diagnosing failures in collaborative systems where compliance depends on the joint behavior of multiple autonomous agents, building on a corrected and reformulated syntax and semantics of our earlier work [KASL23]. The central technical novelty lies in treating collaboration itself as a semantic object. Unlike traditional runtime verification and temporal logic frameworks, which evaluate compliance solely in terms of realized events, the framework developed here distinguishes between what each agent attempted, what was obstructed, and what jointly succeeded. This distinction is essential for responsibility attribution in collaborative contracts and cannot be reconstructed from standard trace semantics after the fact. By formalizing interaction as a collaborative process rather than a zero-sum game, we establish a rigorous basis for blame attribution when contractual obligations are not met. This foundation supports the development of a logic and a runtime verification pipeline capable of handling temporal constraints, reparable violations, and the open-ended structure of real-cases of legal multi-party contracts.

4.1 Overview

In this section, we outline the theoretical and practical necessity for a blame-aware monitoring framework. We begin by motivating the problem through the lens of open-ended normative interactions, where simple binary verdicts of satisfaction or violation are insufficient for dispute resolution. We then detail the specific research questions that guide our inquiry and describe the constructive methodology used to answer them. Finally, we summarize the chapter's core contributions, including the novel Attempt-Based Interaction Model and the Two-Agents Collaborative Normative Logic (TACNL), which collectively enable precise, automated accountability in digital contracts.

4.1.1 Motivation

Normative specifications are typically presented as constraints on what an individual agent must do or avoid. In many practical contexts, however, compliance is not achieved

solely by individual agents but results from interactions where parties may enable, disregard, or obstruct each other. This chapter examines collaborative normative settings where, even without zero-sum competition, strategic behavior and conflicting incentives arise. Our analysis focuses on two agents, as the fundamental challenges of collaboration are fully present even in bilateral contracts.

Interactions and collaboration. Standard deontic formalism describes duties in isolation, yet many norms require that a desired outcome be realized through the joint contributions of both parties. For instance, a tenant's payment succeeds only if the landlord provides a channel to receive it; similarly, occupancy requires both the tenant's attempt to occupy and the landlord's permission. A robust model must therefore distinguish between the *attempts* made by each agent and the *successful collaboration* resulting from their compatibility. Additionally, the model must account for *non-interference*, ensuring that compliance involves refraining from actions that obstruct the counterparty.

Open-ended normative interactions. Contracts often govern long-term collaborations that persist indefinitely unless terminated. A rental agreement, for example, imposes recurring obligations and conditional repairs that repeat over time. This motivates the development of a trace model that supports aggregation at the granularity of the norm (e.g., monthly periods) and allows for *prefix-based evaluation*, as monitoring depends on partial observations of ongoing relationships.

Blame and accountability. When a collaborative norm is breached, it is insufficient to merely state that a violation occurred; it is necessary to identify *who* is responsible and provide justification. This is critical in disputes where enforcement depends on explicit accounts of causation. For example, courts must reconstruct timelines to determine if a landlord's obstruction justified a tenant's non-payment. Furthermore, practical analysis requires a *quantitative assessment* of non-compliance—counting unpaid months or denied access—rather than a simple binary verdict. A blame concept suitable for compliance tools must therefore be formally defined, operationally verifiable, and robust under incremental observation.

Research questions. These considerations motivate the following questions from a formal methods' perspective:

1. How can collaboration be modeled as a first-class semantic object, distinguishing individual attempts from jointly successful outcomes?
2. Which trace abstractions effectively capture cooperative behavior and non-interference at the granularity of contractual periods?

3. How can open-ended clauses, specifically *repetitions* of duties be specified and monitored within a contractual framework?
4. How can blame and quantitative accountability be defined to reflect legal reasoning and support automated monitoring?

4.1.2 Methodology

To address these questions, we adopt a *constructive formal methods* approach. Our methodology is predicated on the rigorous separation of *specification* (the denotational meaning) from *implementation* (the operational mechanism), ensuring tools are correct-by-construction. The research follows a four-phase structural pipeline:

Phase 1: Semantic Abstraction. We abstract the domain of interaction by rejecting atomic trace models in favor of a *multi-layered* approach. Interaction is modeled as the semantic intersection of independent strategies, allowing us to reason formally about obstruction and interference.

Phase 2: Denotational Specification. We adopt a *semantics-first* strategy, defining abstract truth conditions before execution mechanisms. We prioritize *tightness*—the ability to render precise verdicts on finite prefixes. We start with binary compliance, refine it into a blame-assignment model, and extend it to the quantitative domain.

Phase 3: Structural Synthesis. We bridge theory and practice using *automata-theoretic synthesis*. We treat the logic as a blueprint for generating deterministic Moore machines, employing structural induction to ensure the monitor's structure mirrors the contract.

Phase 4: Formal Verification. We validate the pipeline through *conformance proofs*. We prove that the synthesized automata are logically isomorphic to the denotational semantics, guaranteeing that the executable artifacts faithfully represent the normative intent.

4.1.3 Contributions

This chapter establishes a comprehensive framework for the specification and runtime verification of collaborative normative systems. The primary contribution is a verified pipeline bridging high-level normative abstractions and low-level executable monitors. The specific contributions are:

Con1 **Attempt-Based Interaction Model.** We introduce a formal algebra that distinguishes an agent's *attempt* from the *interaction outcome*. Unlike models that conflate failure to act with blocked action, we treat non-interference as a first-class semantic property, enabling the verification of "active participation".

Con2 **The Two-Agents Collaborative Normative Logic (**TACNL**).** We formalize TACNL, a logic tailored for bilateral, periodic contracts. It extends standard deontic specifications with atomic responsibility modalities, regular expression control flow for triggers and guards, and explicit operators for repetition and contrary-to-duty reparations.

Con3 **Theory of Tight and Quantitative Semantics.** We introduce a *Forward Tight Semantics* for online monitoring. This includes a 5-valued verdict domain that distinguishes eager violations from post-violation extensions and a *Quantitative Blame Semantics* that accumulates penalties over time for open-ended contracts.

Con4 **Correct-by-Construction Monitor Synthesis.** We provide a compilation algorithm that transforms TACNL specifications into deterministic Moore machines. We provide structural proofs that this monitor is operationally equivalent to the tight semantics, guaranteeing correctness by synthesis.

Con5 **Formalization of Accountability.** We contribute a formal definition of *blame assignment* in cooperative settings. We prove that our blame semantics is a consistency-preserving refinement of standard violation semantics, providing a mathematical basis for liability assessment.

Organization. We first ground the problem in a rental-agreement example and extract the modeling requirements. We then develop the trace and synchronization layer for collaborative interaction, introduce TACNL, and finally present the monitor-based semantics, blame attribution, and quantitative extensions.

4.2 Use Case: Flat Rental Contract

To ground the theoretical development in a concrete setting, we examine a simplified residential flat rental agreement. This agreement is abstracted to exclude legal boilerplate,

retaining only the core structural elements relevant to our analysis. This abstraction isolates the essential logical phenomena, specifically the collaborative interactions between agents and the temporal evolution of their duties.

Example 42. *We present the structure of this simplified rental contract in Figure Figure 4.1, involving a tenant (agent 1) and a landlord (agent 2). The clauses illustrate cooperative actions such as paying rent and granting occupancy: payment requires both the tenant's offer and the landlord's acceptance, while occupancy requires both the tenant's willingness to occupy and the landlord's permission. The contract also includes a reparation clause and a termination condition, all of which are regulated on a monthly basis and repeat over time.*

Occupancy and Rent Payment

(C1) The Tenant shall pay the full monthly rent on or before the due date.

(C2) The Landlord shall guarantee the Tenant's right to quiet enjoyment and occupancy of the premises, provided the Tenant complies with the agreement.

(C3) In the event of non-payment or late payment of rent:

- (i) If the Tenant fails to pay by the due date, a late fee of 10% of the monthly rent shall be assessed.
- (ii) This late fee shall be due and payable along with the next month's rent.
- (iii) The late fee does not waive any other rights or remedies available to the Landlord under this agreement or the law.

(C4) If the Tenant submits a formal request for necessary repairs, the Landlord shall carry out the required repairs within one (1) month of receiving the request.

Termination Notice and Continued Occupancy

(C5) The Tenant may terminate this rental agreement by giving written notice at least three (3) months prior to the intended termination date. Upon doing so, the Tenant shall:

- (i) Continue to pay the full monthly rent during the three-month notice period.
- (ii) Retain the right to occupy the premises for the entirety of the notice period.
- (iii) Comply with all other terms of the rental agreement during the notice period.
- (iv) Vacate the premises no later than the final day of the notice period, unless otherwise agreed in writing.

Figure 4.1: Use Case: Simplified Rental Contract between a Landlord and a Tenant

4.2.1 Illustrative Interaction Scenarios

To illustrate the practical implications of these clauses, we analyze three hypothetical execution traces over a timeline of months $t = 1, 2, \ldots$. These scenarios highlight the divergence between simple observation (what happened) and normative analysis (who is responsible).

Scenario 1: Obstruction (The "Handshake" Problem). Consider the obligation to pay rent (Clause C1) at month $t = 1$.

- **Action:** The Tenant attempts to initiate the bank transfer.
- **Reaction:** The Landlord, wishing to manufacture a reason for eviction, closes the receiving bank account or refuses the check.
- **Observable Outcome:** No money is transferred.
- **Normative Analysis:** A naive monitor observing only "payment status" would flag a violation by the Tenant. However, in a collaborative framework, the Tenant has fulfilled their duty by *attempting* the act. The failure is attributed to the Landlord's interference, necessitating a distinction between *attempted payment* and *successful payment*.

Scenario 2: Violation and Reparation (The CTD Chain). Consider a scenario where the Tenant genuinely runs out of funds.

- **Month 1:** Tenant fails to pay rent.
- **Immediate Verdict:** Violation of C1. The contract enters a "violated" state.
- **Normative Shift:** Clause C3 (Reparation) activates. The Tenant now holds a *secondary obligation*: pay Month 1 Rent + Month 2 Rent + 10% Fee at Month 2.
- **Month 2:** Tenant pays the total accumulated amount.
- **Final Verdict:** The violation is considered *repaired*, and the contract returns to a compliant state. The monitor must track this "state recovery" rather than permanently aborting at the first failure.

Scenario 3: Termination and Temporal Reconfiguration. Consider the Tenant exercising the power to terminate (Clause C5).

- **Month 5:** Tenant issues a formal written notice to terminate.
- **Normative Shift:** This is not a violation but a *state change*. The contract, previously open-ended, is now fixed to terminate at Month 8 ($5+3$).
- **Month 6–8:** The Tenant stops paying rent, assuming the notice ended the contract immediately.
- **Verdict:** Violation. The monitor must enforce that the *obligations persist* during the notice period (Clause C5-i). The exercising of power did not delete the obligations but merely set a horizon for their expiration.

4.2.2 Formal Modeling Challenges

While conceptually straightforward, formalizing this contract requires resolving several semantic challenges that standard specification languages (such as LTL or standard Deontic Logic) fail to address adequately.

1. The "Handshake" Semantics (Attempts vs. Outcomes). Clauses C1 and C2 illustrate that compliance is a *joint achievement*. For example, "Rent Payment" (C1) semantically implies that the Tenant offers the money *and* the Landlord accepts it. Standard trace models often represent this as a single atomic event *pay*. However, in a dispute, we must distinguish between the Tenant failing to offer rent (Tenant violation) and the Tenant offering but the Landlord blocking the transfer (Landlord interference). **Challenge:** The formalism must treat *attempts* and *successful outcomes* as distinct semantic layers to correctly attribute blame.

2. Contrary-to-Duty Structures (Reparation). Clause C3 introduces non-monotonicity. If C1 is violated, the contract does not simply evaluate to *false* (breach). Instead, it transitions to a sub-ideal state where a secondary obligation (the late fee) becomes active. This creates a *reparation chain*: Primary Duty $\rightarrow$ Violation $\rightarrow$ Secondary Duty. **Challenge:** The logic must support *recoverable violations*, allowing the monitoring status to fluctuate between "violated" and "repaired" over time.

3. Reactive Triggers and Monitoring Horizons. Clause C4 (Repairs) imposes a duty with a relative deadline: "within one month of request". From a monitoring perspective, this requires *forward-looking tightness*. We cannot wait for an infinite trace to determine if the repair happened. **Challenge:** The semantics must identify the *exact finite prefix* where the deadline is missed to trigger a verdict immediately (eager rejection).

4. Hohfeldian Powers and Structural Change. Clause C5 (Termination) represents a *Power* rather than a simple duty. By exercising this power, the Tenant unilaterally alters the normative landscape, changing an open-ended/infinite contract into a finite one (ending in 3 months). **Challenge:** The framework must model agents not only as actors within the rules but as entities capable of modifying the rules or their duration dynamically.

Summary

These scenarios motivate the need for a new logic of collaboration. This logic must represent strategies of multiple agents acting in tandem rather than in opposition, express obligations and powers in a uniform framework, capture open-ended dynamics, and support the analysis of compliance and blame. Such a framework requires both a formal model of interacting agent behaviors and a language with syntax and semantics suited to normative reasoning in collaborative, time-sensitive contracts.

4.3 Abstractions for Collaborative Interaction

This section builds on the trace models and synchronization operators introduced in Section 2.2. We keep periodic synchronized set traces as the underlying execution model and define an interaction layer that (i) distinguishes agent-local attempts from jointly successful outcomes and (ii) prepares the ground for strategy analysis and, later, normative evaluation.

The choice of synchronization is not cosmetic. It clarifies what constitutes feasible joint behavior, which, in turn, drives semantic validity, strategy quantification, and downstream analysis of compliance, responsibility, or blame.

We introduce the attempted/successful abstraction, suited for capturing interference and collaboration between agents in the context of collaborative normative specifications, and discuss how it resembles and differs from existing synchronization techniques in the literature. This contribution is not cosmetic: it determines which joint behaviors are feasible, what formulas are satisfied, and how knowledge and responsibility are assessed.

Method and structure. To reason about *attempted, prevented, and successful collaboration*, we introduce our contribution of attempted/successful abstraction on periodic, agent-tagged set traces of collaborative alphabets: (i) *attempted* collaboration is captured by per-agent enabler sets (active participation) in a period; (ii) *prevented* collaboration arises from blocker (non-interference violations); and (iii) *successful* collaboration is defined by the operator Succ, the point-wise intersection of the two agents' untagged suggested sets (equivalently, the set-theoretic image of lockstep-with-handshakes on the collaboration alphabet).

Modeling Requirements

The aim is to capture two complementary facets of collaboration between agents that standard system-centric models overlook, as illustrated in Example 42:

- **Negative performance (non-interference).** One agent must *refrain* from actions that would block the other from achieving a legitimate objective. In a tenant–landlord scenario, the landlord should not perform blocking actions such as cut_power, change_lock, or enter_flat, which would hinder the tenant's ability to occupy the flat.
- **Active participation (positive performance).** An agent must *contribute* and perform actions that make the objective of the counterpart achievable within the agreed period. For example, the landlord should provide/confirm account_info, ack_pay, and grant_occ upon settlement; symmetrically, the tenant should pay_rent and not return_payment.

These performance aspects are common in normative / contract settings but are underexplored by classical synchronization operators (interleaving, lockstep), which specify *how* events align rather than whether agents *refrain* from harmful actions or *contribute* enabling ones.

4.3.1 From Metric-Timed to Collaborative Periodic Synchronized Set Traces

In this subsection, we make the abstraction process explicit by structuring it into a 2-steps transformation. We start from raw timed observations that record what each agent did and when. These observations are first organized into agent-local metric-timed traces, preserving responsibility and temporal order. We then impose the contractual granularity by aggregating behavior into period-based set traces aligned with the contract calendar.

Finally, concrete actions are abstracted into collaborative objectives, yielding a representation in which attempts, obstruction, and potential joint success can be analyzed independently. Each step reduces representational detail while preserving all distinctions that are normatively relevant. No behavior is discarded or reinterpreted; it is only lifted to a level suitable for contractual reasoning. Example 43 serves as the starting point of this pipeline, grounding the abstractions in a concrete four-month execution before any synchronization or success computation is applied.

Example 43. *Four-month rental dispute as metric-timed traces We return to the rental agreement introduced in Example 42, in which a tenant (agent 1) and a landlord (agent 2) have signed a monthly contract governing rent payments and occupancy. We observe their concrete behavior over a period of four months, corresponding to days* $t \in \{0, \ldots, 120\}$.

*During the first month of observation, the tenant transfers the rent on day 5 (*pay_rent*), yet does not move into the flat. One day later, on day 6, the landlord formally acknowledges receipt of the payment (*ack_pay*).*

*In the second month, the interaction becomes more active. On day 35, the tenant begins occupying the flat (*start_occ*) and then makes the rent payment the next day. The landlord responds by issuing an administrative acknowledgement on day 37 (*ack_pay*).*

In the third month, the dispute begins: the tenant continues to occupy the flat but makes no rent payments, while the landlord remains passive and takes no observable action.

*In the fourth month, the tenant attempts to address the earlier non-payment by transferring a late fee on day 95 (*pay_rent_f*). This attempt is immediately rejected by the landlord on the same day (*ref_pay*). On the following day, day 96, the landlord escalates the situation by changing the lock, thereby actively preventing continued occupancy (*change_lock*). From the concrete observations, we identify the following agent-local action alphabets:*

$$\Sigma_1 = \{\mathit{pay\_rent},\ \mathit{pay\_rent\_f},\ \mathit{start\_occ},\ \mathit{stop\_occupy}\},$$
$$\Sigma_2 = \{\mathit{ack\_pay},\ \mathit{grant\_occ},\ \mathit{ref\_pay},\ \mathit{change\_lock}\}.$$

The observed behavior of each agent is encoded as a metric-timed trace, where each event is paired with its occurrence time:

$$\tau_1^{\mathrm{mt}} = \langle(\mathit{pay\_rent}, 5),\ (\mathit{start\_occ}, 35),\ (\mathit{pay\_rent}, 36), (\mathit{pay\_rent\_f}, 95)\rangle,$$
$$\tau_2^{\mathrm{mt}} = \langle(\mathit{grant\_occ}, 1),\ (\mathit{ack\_pay}, 6),\ (\mathit{ack\_pay}, 37),\ (\mathit{ref\_pay}, 95),\ (\mathit{change\_lock}, 96)\rangle.$$

These timed words constitute the most concrete representation of the interaction and serve as the starting point for the abstraction pipeline developed in this section.

From these metric-timed traces, the behavior in each month is precise at a day level and could be abstracted in a monthly view. In the first month, rent is paid and accepted, and occupancy is granted but not yet exercised. In the second month, occupancy begins, and payment is acknowledged without conflict. In the third month, the tenant continues to occupy while failing to pay, and the landlord remains passive. In the fourth month, the tenant attempts to remedy the earlier violation by paying a late fee, which is explicitly rejected, and then engages in active obstruction of occupancy by changing the lock. While this description is clear at the level of concrete events, it is not yet aligned with the contractual semantics, which operate on monthly obligations and joint outcomes rather than individual timestamps. This motivates the next step: synchronizing the agent-local traces into a common period-based structure that supports collaboration, interference, and normative evaluation.

Step 1: Abstraction into Periodic Synchronized Set trace

Since the contract regulates obligations and permissions *per month* rather than per day, we periodize the timeline into month windows $I_1, I_2, I_3, \ldots$ and aggregate the actions of each agent within the same window into a *set*. This yields a synchronous round-based abstraction aligned with the contract calendar.

Definition 32 (Periodic synchronized set trace)**.** *A* periodic synchronized set trace *for agent i over the alphabet Σ_i, relative to a global clock with a fixed period length, is an infinite sequence of action sets indexed from* 1*:*

$$\pi_i \;=\; \langle A_1^{(i)},\, A_2^{(i)},\, A_3^{(i)},\, \ldots \rangle \;\in\; (2^{\Sigma_i})^{\infty}.$$

Each element $A_k^{(i)} \subseteq \Sigma_i$ represents the set of actions attributed to agent i during period k (where $\emptyset$ denotes inactivity).

Having defined the target domain of period-based set traces, we now establish the link to the concrete observations. Since our starting point are metric-time traces of individual events (as seen in Example 43), we must map these timestamps into the discrete calendar defined by the contract. The following definition formalizes this abstraction process by partitioning the timeline into fixed-size windows and aggregating all events occurring within each window into a single set.

Definition 33 (Aggregation to periodic synchronized set trace)**.** *Let $x \in \mathbb{N}$ be a fixed period length. Define the aggregation function*

$$\mathsf{AGG}_x : (\Sigma \times \mathbb{N})^* \longrightarrow (2^{\Sigma})^*$$

as follows.

Given a finite metric-timed trace

$$\tau^{\mathrm{mt}} = \langle (a_1,t_1),\ldots,(a_n,t_n)\rangle \quad \textit{with } a_j \in \Sigma,\ t_j \in \mathbb{N} \textit{ for all } j \in \{1,\ldots,n\},$$

we partition time into period windows of length x:

$$I_k \ := \ [(k-1)x+1,\ kx] \qquad (k \geq 1).$$

The aggregated periodic synchronized set trace

$$\mathsf{AGG}_x(\tau^{\mathrm{mt}}) := \pi = \langle A_1,\ldots,A_K\rangle$$

is defined by

$$A_k := \{\, a \in \Sigma \mid \exists t \in I_k :\ (a,t) \in \tau^{\mathrm{mt}} \,\},$$

where

$$K := \max\{\, k \mid I_k \cap \{t_1,\ldots,t_n\} \neq \emptyset \,\}.$$

Proof. By definition of AGG_x, membership $a \in A_k^{(i)}$ and $b \in A_\ell^{(i)}$ yields witnesses $t_a \in I_k$ and $t_b \in I_\ell$ with $(a,t_a),(b,t_b) \in \tau_i^{\mathrm{mt}}$. Since $k < \ell$, the windows are disjoint and strictly ordered (i.e., $\max(I_k) < \min(I_\ell)$), which implies $t_a < t_b$. □

Lemma 26 (Order preservation under aggregation)**.** *Let $\tau_i^{\mathrm{mt}} \in (\Sigma_i \times \mathbb{N})^*$ be a finite metric-timed trace and let $x \in \mathbb{N}$. Let*

$$\pi_i := \mathsf{AGG}_x(\tau_i^{\mathrm{mt}}) = \langle A_1^{(i)},\ldots,A_K^{(i)}\rangle.$$

Then aggregation preserves inter-period order in the following sense:

$$\forall k,\ell \in \{1,\ldots,K\},\ k < \ell,\ \forall a \in A_k^{(i)},\ \forall b \in A_\ell^{(i)},\ \forall t_a,t_b :$$
$$\big((a,t_a) \in \tau_i^{\mathrm{mt}} \wedge t_a \in I_k\big) \ \wedge \ \big((b,t_b) \in \tau_i^{\mathrm{mt}} \wedge t_b \in I_\ell\big) \ \Rightarrow \ t_a < t_b.$$

Proof. By definition of AGG_x, membership $a \in A_k^{(i)}$ and $b \in A_\ell^{(i)}$ yields witnesses $t_a \in I_k$ and $t_b \in I_\ell$ with $(a,t_a),(b,t_b) \in \tau_i^{\mathrm{mt}}$. Since $k < \ell$ implies $I_k < I_\ell$ by construction of the period windows, we obtain $t_a < t_b$. □

Nevertheless, aggregation has a cost, as it loses intra-period order, action multiplicity, and exact timestamps. Therefore, selecting the appropriate period normalization is crucial.

Example 44 (Transforming timed words into periodic set traces, continued from Example 43)**.** *By fixing the period to 30, we can decompose the 120 days into 4 horizons:*

$$I_1 = [1,30], \qquad I_2 = [31,60], \qquad I_3 = [61,90], \qquad I_4 = [91,120].$$

The two timed traces τ_1^{mt} *and* τ_2^{mt} *are transformed using the function* AGG_{30} *transformation respectively into:*

$$\pi_1 = \langle \underbrace{\{\mathit{pay\_rent}\}}_{I_1}, \underbrace{\{\mathit{start\_occ},\ \mathit{pay\_rent}\}}_{I_2}, \underbrace{\emptyset}_{I_3}, \underbrace{\{\mathit{pay\_rent\_f}\}}_{I_4} \rangle,$$

$$\pi_2 = \langle \underbrace{\{\mathit{grant\_occ},\ \mathit{ack\_pay}\}}_{I_1}, \underbrace{\{\mathit{ack\_pay}\}}_{I_2}, \underbrace{\emptyset}_{I_3}, \underbrace{\{\mathit{ref\_pay\_f},\ \mathit{change\_lock}\}}_{I_4} \rangle.$$

By Lemma 26, for agent 1 the element $\mathit{start\_occ} \in A_2^{(1)}$ *precedes* $\mathit{pay\_rent\_f} \in A_4^{(1)}$ *(witness times* $35 < 95$*), and similarly for agent 2.*

Step 2: Abstraction to a Collaboration Alphabet

The second aspect of the abstraction is the notion of interaction alphabets. The action alphabets of both agents are merged into a single alphabet, and the trace encodes that when a is an event, it indicates that an agent has either instantiated it or accepted to collaborate on it. Conversely, the absence of an action in the event at Set I_k indicates that the agent either did not perform it or actively performed a blocking action to prevent it. This abstraction is demonstrated using the running example, showing that it maintains the same meaning while reducing the number of action types. An operator is then introduced to compute the successful interaction at each period of the synchronized trace. This transformation cannot be automated unless the set of enabling and interfering actions is explicitly stated in the normative specification; otherwise, the digitization engineer must define them manually, as is done here.

We consider three collaborative objectives that are identified in the Example 42: PAY_R (rent payment), PAY_F (penalty/late-fee payment), and OCC (tenant's occupancy). We write $\Sigma_C = \{\mathsf{PAY\_R}, \mathsf{PAY\_F}, \mathsf{OCC}\}$.

In the next step, we need to define how the actions of the agents, namely Σ_1 and Σ_2, relate to collaborative action over Σ_C, more specifically, whether they are enabler or interference actions. We do not take the union $\Sigma_1 \cup \Sigma_2$ as a synchronization operator discussed in the Subsection 2.2.6. Instead, we define a many-to-one *abstraction* from concrete per-agent actions to the collaboration alphabet Σ_C. On the tenant side, pay_rent and pay_rent_f are enablers because they instantiate the tenant's contribution toward PAY_R and PAY_F; start_occ is an enabler for OCC because it is the tenant's side of taking possession. A chargeback return_payment and return_payment_f are blocking: they nullify the very

transfer that PAY_R and *PAY_F* rely on. On the landlord side, ack_pay enables PAY_R, while grant_occ enables OCC by authorizing access. In contrast, ref_pay blocks PAY_R even if the tenant initiated payment, and change_lock, cut_power, or enter_flat block OCC by making continued possession impracticable or unlawful. This enabler/blocker partition is precisely what our "suggested/successful" abstraction needs: success in a period occurs when both sides propose the required enablers and neither side performs a blocker.

Distinguishing enabling from Blocking actions For each agent $i \in \{1,2\}$ we partition the alphabet into Σ_i^A (actions that constitute *active participation*, i.e., enablers) and Σ_i^I (actions that constitute *negative performance*, i.e., blockers), with $\Sigma_i = \Sigma_i^A \cup \Sigma_i^I$, assuming that they are disjoint. In our running example:

$$\Sigma_1^A = \{\,\mathsf{pay\_rent},\ \mathsf{pay\_rent\_f},\ \mathsf{start\_occ}\,\},$$
$$\Sigma_1^I = \{\,\mathsf{refuse\_inspection}\,\}.$$
$$\Sigma_2^A = \{\,\mathsf{ack\_pay},\ \mathsf{grant\_occ}\,\},$$
$$\Sigma_2^I = \{\,\mathsf{ref\_pay},\ \mathsf{change\_lock},\ \mathsf{ref\_pay\_f},\ \mathsf{enter\_flat}\,\}.$$

with:

$$\Sigma_1 = \Sigma_1^A \uplus \Sigma_1^I, \qquad \Sigma_2 = \Sigma_2^A \uplus \Sigma_2^I, \qquad \Sigma = \Sigma_1 \cup \Sigma_2.$$

For brevity, below we write the *enabler* and *blocker* sets as

$$E_i := \Sigma_i^A \quad \text{and} \quad B_i := \Sigma_i^I \qquad (i \in \{1,2\}).$$

Presence of $a \in E_i$ in period k signals that agent i took a *positively contributing* action; presence of $b \in B_i$ signals a *defeating* (interfering) action. The absence of a symbol indicates it was not suggested/endorsed during that period.

Transformation sketch After defining this relation, we can transform any periodic synchronous word π_1 over Σ_1 and π_2 over Σ_2 to a corresponding word over Σ_C, written π_i^C:

- If an action a is on event A from π_i and that action is in E_i, then this action is transformed to its equivalent collaborative action and inserted on the resulting word π_i^C.
- If an action a is on event A from π_i and that action is in B_i, then this action is not transformed and not inserted on the resulting word π_i^C.

And additionally, we do not add any collaborative action not present in an event on the equivalent event in π_i^C unless it is a continuous collaboration with implicit collaboration, as start_occ signals the start of occupying the flat, so it is kept inserted in the following events in the timed word of the tenant and landlord as long as tenant does not leave nor the landlord blocks actively the occupation.

Example 45 (Transforming the periodic synchronized words over Σ_1, Σ_2 to periodic synchronized words over Σ_C)**.** *Using the traces from Example 44,*

$$\pi_1 = \langle \underbrace{\{pay\_rent\}}_{I_1}, \underbrace{\{start\_occ, pay\_rent\}}_{I_2}, \underbrace{\emptyset}_{I_3}, \underbrace{\{pay\_rent\_f\}}_{I_4} \rangle,$$
$$\pi_2 = \langle \underbrace{\{grant\_occ,\ ack\_pay\}}_{I_1}, \underbrace{\{ack\_pay\}}_{I_2}, \underbrace{\emptyset}_{I_3}, \underbrace{\{ref\_pay\_f,\ change\_lock\}}_{I_4} \rangle.$$

With $E_1(\mathsf{PAY\_R}) = \{pay\_rent\}$,$E_1(\mathsf{PAY\_F}) = \{pay\_rent\_f\}$ $E_2(\mathsf{PAY\_R}) = \{ack\_pay\}$, $B_2(\mathsf{PAY\_R}) = \{ref\_pay\}$, $E_1(\mathsf{OCC}) = \{start\_occ\}$, $E_2(\mathsf{OCC}) = \{grant\_occ\}$, $B_2(\mathsf{OCC}) = \{change\_lock\}$, *Consequently, the collaboration alphabet is:*

$$\Sigma_C = \{\mathsf{PAY\_R},\ \mathsf{PAY\_F},\ \mathsf{OCC}\}.$$

The corresponding collaboration-trace abstractions of π_1 *and* π_2 *are* π_1^C *and* π_2^C*:*

$$\pi_1^C = \langle \underbrace{\{\mathsf{PAY\_R}^{(1)}\}}_{I_1}, \underbrace{\{\mathsf{OCC}^{(1)},\ \mathsf{PAY\_R}^{(1)}\}}_{I_2}, \underbrace{\{\mathsf{OCC}^{(1)}\}}_{I_3}, \underbrace{\{\mathsf{PAY\_F}^{(1)}\}}_{I_4} \rangle,$$
$$\pi_2^C = \langle \underbrace{\{\mathsf{OCC}^{(2)},\ \mathsf{PAY\_R}^{(2)}\}}_{I_1}, \underbrace{\{\mathsf{OCC}^{(2)},\ \mathsf{PAY\_R}^{(2)}\}}_{I_2}, \underbrace{\{\mathsf{OCC}^{(2)}\}}_{I_3}, \underbrace{\emptyset}_{I_4} \rangle.$$

Reading: in month I_1*, agent 1 positively contributes to* PAY_R*, and agent 2 allows* OCC *and accepts* PAY_R*. In* I_2*, agent 1 pays the rent and occupies the flat and agent 2 contributes to* PAY_R *and allows the occupation* OCC*. In* I_3*, agent 1 keeps* OCC *but does not pay, and the landlord keeps allowing* OCC*. In* I_4*, the landlord blocks both* PAY_R *and denies* OCC*, although the tenant wants to keep occupying the flat and pays.*

Optional Step: Successful Collaboration Trace

The abstraction introduced so far separates agent-local attempts and interference while preserving all information required for responsibility analysis. In some situations, however, it is useful to additionally extract the *jointly successful* outcomes that emerge when both agents contribute compatibly within the same contractual period. We introduce an optional transformation that computes such outcomes explicitly. The construction follows the same underlying principle as lockstep synchronization with handshakes inspired from

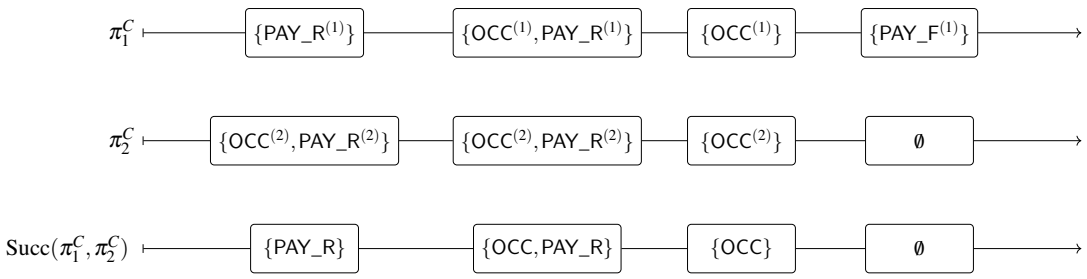


Figure 4.2: Example: successful collaboration computation example

Definition 15: a collaborative action is realized at a given period if and only if both agents propose it in that period. Technically, this is captured by erasing agent tags and intersecting the per-period action sets. Unlike earlier abstraction steps, this operator is not used as the primary semantic basis of the framework, since it intentionally discards information about obstruction and responsibility. Instead, it serves as a derived view that characterizes successful collaboration in isolation, making precise the connection between our attempt-based abstraction and classical handshake-based synchronization models.

Definition 34 (Successful collaboration operator)**.** *Let* Σ *be an alphabet and let* $\Sigma^{(1)} := \{a^{(1)} \mid a \in \Sigma\}$ *and* $\Sigma^{(2)} := \{a^{(2)} \mid a \in \Sigma\}$ *be disjoint tagged copies. The* successful collaboration operator*, written* Succ *of two periodic synchronous traces* π_1 *over* $\Sigma^{(1)}$ *and* π_2 *over* $\Sigma^{(2)}$ *on the same maximum horizon* T *is defined event-wise by:*

$$\forall k \in \{1,\dots,T\}, \mathrm{Succ}(\pi_1,\pi_2)[k] \ := \mathsf{unlab}\big(\pi_1[k]\big) \cap \mathsf{unlab}\big(\pi_2[k]\big).$$

where the function unlab *is removing any of the agent identifiers tag from an event, e.g.* $\mathsf{unlab}(\{\mathsf{OCC}^2\}) = \{\mathsf{OCC}\}$.

Example 46 (Successful collaboration)**.** *From the two transformed traces* π_1^C *and* π_2^C *from the Example 45 We illustrate* $\mathrm{Succ}(\pi_1^C, \pi_2^C)$ *in Figure 4.2*

Basic properties. Since Succ is pointwise set intersection after tag erasure, it inherits three immediate facts: *(i) Commutative and idempotent*: $\mathrm{Succ}(\pi_1,\pi_2) = \mathrm{Succ}(\pi_2,\pi_1)$ and $\mathrm{Succ}(\pi,\pi) = \mathsf{unlab}(\pi)$. *(ii) Monotone (pointwise* $\subseteq$*)*: writing $\pi \le \pi'$ to mean $\forall k$: $\mathsf{unlab}(\pi[k]) \subseteq \mathsf{unlab}(\pi'[k])$, if $\pi_1 \le \pi_1'$ and $\pi_2 \le \pi_2'$ then $\mathrm{Succ}(\pi_1,\pi_2) \le \mathrm{Succ}(\pi_1',\pi_2')$. *(iii) Absorbing empty trace*: if $\forall k,\ \pi_2[k] = \emptyset$, then $\forall k,\ \mathrm{Succ}(\pi_1,\pi_2)[k] = \emptyset$.

Remark 6. *Relation to* $\|^A_{\mathrm{hs}}$*. If we declare every collaborative objective to be a handshake,* $A = \Sigma_C$*, then the lockstep-with-handshakes product enforces that* PAY_R, PAY_F, OCC *can only appear at a period when both agents present the same letter. Concretely, if we view each period's set* $A_k^{(i)}$ *as the multiset of letters occurring "at that round" and apply*

$\|^{A}_{\mathrm{hs}}$ *round-wise, then after collapsing paired handshakes to a single shared symbol (via* coll_A*) and unlabeling, we obtain exactly the success:*

$$\mathrm{Succ}(\pi_1, \pi_2) \;=\; \mathsf{unlab}\big(\mathsf{coll}_{\Sigma_C}\big(\pi_1 \,\|^{\Sigma_C}_{\mathrm{hs}}\, \pi_2\big)\big),$$

i.e., Succ *is the set-theoretic intersection semantics of lockstep-plus-handshakes on the collaboration alphabet.*

However, this abstraction masks the specific source of interference, effectively hiding the agent responsible for the failure. As our framework aims to explicitly assign blame, we introduce this operator primarily as an optional view to characterize joint success. We now move to reasoning about strategies from agents, where each agent makes a plan based on the other agent's past behavior.

4.3.2 Interaction Strategies as Moore Machines

Most monitoring approaches analyse compliance only after an execution has occurred. This chapter goes further by modelling how agents plan their interaction in anticipation of each other's behavior. By representing strategies as Moore machines, we make responsibility analyzable as an ex ante property of interacting plans, not merely as a retrospective judgement over completed traces.

Strategy Model

We capture interaction strategies with *input/output* models that operate at the period granularity. At each period k, each agent i observes the other agent's period-k output and updates its internal state to produce its own period-$(k{+}1)$ output. We use *Moore machines* so that an agent's output at period k depends only on its current state (perfect monitoring with one-period delay). A synchronous feedback composition couples the two machines.

Definition 35 (Deterministic Moore machine)**.** *A deterministic Moore machine for agent i is a 6-tuple*

$$M_i \;=\; (S_i, s_i^0,\, \Sigma_I^i,\, \Sigma_O^i,\, \delta_i,\, \lambda_i),$$

where:

- S_i *is a finite set of states with initial state* $s_i^0 \in S_i$*;*
- Σ_I *is the input alphabet (the other agent's* untagged *collaborative letters);*

- Σ_O^i *is the set of output symbols;*
- $\delta_i : S_i \times 2^{\Sigma_I} \to S_i$ *is a deterministic transition function;*
- $\lambda_i : S_i \to 2^{\Sigma_O}$ *is the output function.*

Given an input stream $X = (X_0, X_1, \dots)$ *with* $X_k \subseteq \Sigma_I$*, the induced run is* $s_i^0, s_i^1, \dots$ *with* $s_i^{k+1} = \delta_i(s_i^k, X_k)$ *and outputs* $Y_k = \lambda_i(s_i^k)$.

Definition 36 (Run and output of a deterministic Moore machine)**.** *Let*

$$M_i \;=\; (S_i, s_i^0,\ \Sigma_I^i,\ \Sigma_O^i,\ \delta_i,\ \lambda_i)$$

be a deterministic Moore machine for agent i as in Definition 35. An input stream *for* M_i *is a finite or infinite sequence* $X = (X_0, X_1, \dots)$ *with* $X_k \subseteq \Sigma_I^i$ *for all positions k.*

Run. *The* run of M_i on X *is the unique sequence of states*

$$\rho_i(X) \;=\; (s_i^0, s_i^1, s_i^2, \dots)$$

inductively defined by

$$s_i^0 := s_i^0, \qquad s_i^{k+1} := \delta_i(s_i^k, X_k) \quad \text{for all } k \geq 1.$$

For finite input X of length $n{+}1$ *we write* $|X| = n{+}1$ *and* $\rho_i(X) = (s_i^0, \dots, s_i^{n+1})$.

Extended transition function. *For later use, we define the extended transition function*

$$\delta_i^* : S_i \times (2^{\Sigma_I^i})^* \to S_i$$

by

$$\delta_i^*(s, \varepsilon) := s, \qquad \delta_i^*(s, X_0 \cdot X') := \delta_i^*\big(\delta_i(s, X_0), X'\big),$$

for $X_0 \in 2^{\Sigma_I^i}$ *and* $X' \in (2^{\Sigma_I^i})^*$*. For a finite input word X, we write*

$$\delta_i(s_i^0, X) \;:=\; \delta_i^*(s_i^0, X),$$

so that the last state of the run on X is $\delta_i(s_i^0, X)$.

Output word and terminal output. *The* output word *induced by* M_i *on X is*

$$\lambda_i^\omega(X) \;:=\; (Y_0, Y_1, \dots) \quad \text{with} \quad Y_k := \lambda_i(s_i^k).$$

For a finite input word X the terminal output *of* M_i *on X is*

$$\lambda_i\big(\delta_i(s_i^0, X)\big),$$

which is the output associated with the last state of the run on X.

Synchronous Feedback Composition

We present the property of two Moore machines that can feed each other and progress together.

Definition 37 (Complementary Moore machines)**.**

Let $M_i = (S_i, s_i^0, \Sigma_I^i, \Sigma_O^i, \delta_i, \lambda_i)$ *and* $M_j = (S_j, s_j^0, \Sigma_I^j, \Sigma_O^j, \delta_j, \lambda_j)$

be two deterministic Moore machines, we say that M_i and M_j are complementary *if and only if:*

$\Sigma_I^i = \Sigma_O^j$ *and* $\Sigma_O^i = \Sigma_I^j$.

In the following, we introduce an example of how to use Moore machines to capture two strategies that the landlord and the tenant should consider in the motivating example.

Example 47 (Interaction strategies and their Moore encodings)**.** *Consider the two first informal strategies* $\mathfrak{S}_1^1$ *and* $\mathfrak{S}_2^1$ *from respectively the tenant(1) and the landlord(2):*

Tenant $\mathfrak{S}_1^1$**.** *"I pay in the first month; from the second month on, I occupy and keep paying as long as the landlord does not stop me. If the landlord stops my occupancy, I stop paying."*

Landlord $\mathfrak{S}_2^1$**.** *"I enable occupancy from the first month and accept payment; if the tenant fails to pay for* two consecutive *months, I stop enabling occupancy." We encode both of those strategies using:* $\Sigma_C^{(1)} := \{a^{(1)} \mid a \in \Sigma_C\}$ *and* $\Sigma_C^{(2)} := \{a^{(2)} \mid a \in \Sigma_C\}$ *be the tagged disjoint copies. Both machines use* $S = \{s_0, s_1, s_2\}$ *with initial state* s_0*. Transitions are guarded by the current letters of the agent* other *(seen as a set).*

The tenant strategy and the landlord strategy are depicted explicitly by their Moore machine encoding in Figure 4.3a and Figure 4.3b, respectively.

$$M_1^1 = (S, s_0, \Sigma_I^{(1)}, \Sigma_O^{(1)}, \delta_1, \lambda_1), \quad \Sigma_I^{(1)} = 2^{\Sigma_C^{(2)}}, \quad \Sigma_O^{(1)} = 2^{\Sigma_C^{(1)}},$$
$$M_2^1 = (S, s_0, \Sigma_I^{(2)}, \Sigma_O^{(2)}, \delta_2, \lambda_2), \quad \Sigma_I^{(2)} = 2^{\Sigma_C^{(1)}}, \quad \Sigma_O^{(2)} = 2^{\Sigma_C^{(2)}}.$$

For $X \subseteq \Sigma_C^{(2)}$, $Y \subseteq \Sigma_C^{(1)}$*:*

$$\delta_1(s_0, X) = \begin{cases} s_1 & \mathsf{OCC}^{(2)} \in X \\ s_0 & \textit{otherwise} \end{cases}, \quad \delta_1(s_1, X) = \begin{cases} s_1 & \mathsf{OCC}^{(2)} \in X \\ s_2 & \textit{otherwise} \end{cases}, \quad \delta_1(s_2, X) = s_2,$$

$$\delta_2(s_0,Y)=\begin{cases}s_0 & \mathsf{PAY\_R}^{(1)}\in Y\\ s_1 & \textit{otherwise}\end{cases},\ \delta_2(s_1,Y)=\begin{cases}s_0 & \mathsf{PAY\_F}^{(1)}\in Y\\ s_2 & \mathsf{PAY\_R}^{(1)}\in Y\\ s_1 & \textit{otherwise}\end{cases},\ \delta_2(s_2,Y)=s_2.$$

Notice that two Moore machines are complementary.

Now we move to the step where both strategies are fixed and transformed to compute their outcome. To do so, we introduce the product of two complementary Moore machines.

Definition 38 (Product of Complementary Deterministic Moore Machines)**.**

Let $M_i = (S_i, s_i^0, \Sigma_I^j, \Sigma_O^i, \delta_i, \lambda_i)$ *and* $M_j = (S_j, s_j^0, \Sigma_I^i, \Sigma_O^j, \delta_j, \lambda_j)$

be two deterministic complementary Moore machines. The product *of* M_i *and* M_j *is the automaton*

$$M_i \otimes M_j = (\Sigma, Q, q_0, \delta, F)$$

where

- $\Sigma = 2^{\Sigma_O^i \cup \Sigma_O^j}$ *is the* joint alphabet*,*
- $Q = S_i \times S_j$ *is the* state space*,*
- $q_0 = (s_i^0, s_j^0)$ *is the* initial state*,*
- $F = \emptyset$ *,*
- $\delta \subseteq Q \times \Sigma \times Q$ *is the* transition relation*, defined by*

  $$\big((s_i,s_j),A,(s_i',s_j')\big)\in\delta$$

  if and only if

  $$A=\lambda_i(s_i)\cup\lambda_j(s_j),\quad s_i'=\delta_i(s_i,\lambda_j(s_j)),\quad s_j'=\delta_j(s_j,\lambda_i(s_i)).$$

The language $L(M_i \otimes M_j) \subseteq \Sigma^\omega$ *consists of all infinite words*

$$\langle A_1, A_2 \ldots \rangle$$

such that there exists a run

$$(s_i^0,s_j^0) \xrightarrow{A_1} (s_{i,1},s_{j,1}) \xrightarrow{A_2} (s_{i,2},s_{j,2}) \xrightarrow{A_3} \ldots$$

with $A_k = \lambda_i(s_{i,k}) \cup \lambda_j(s_{j,k})$ *for all* $k \geq 1$.

Lemma 27 (Unique run and word of product machines)**.** *Let*

$$M_i = (S_i, s_i^0, \Sigma_I^j, \Sigma_O^i, \delta_i, \lambda_i) \quad \textit{and} \quad M_j = (S_j, s_j^0, \Sigma_I^i, \Sigma_O^j, \delta_j, \lambda_j)$$

be two complementary deterministic Moore machines. Then their product $M_i \otimes M_j$ admits a unique run

$$(s_i^0, s_j^0)\,(s_{i,1}, s_{j,1})\,(s_{i,2}, s_{j,2})\ldots$$

and this run induces a unique word

$$\pi \;=\; \langle A_1, A_1, A_2, \ldots\rangle \;\in\; \left(2^{\Sigma_O^i \cup \Sigma_O^j}\right)^{\omega},$$

where

$$A_t \;=\; \lambda_i(s_{i,t}) \cup \lambda_j(s_{j,t}), \qquad t \geq 0,$$

and the successor states are determined by

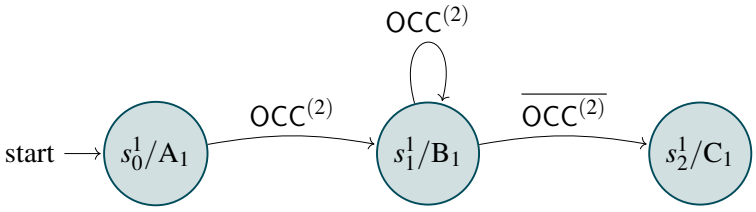
$$s_{i,t+1} = \delta_i(s_{i,t}, \lambda_j(s_{j,t})), \qquad s_{j,t+1} = \delta_j(s_{j,t}, \lambda_i(s_{i,t})).$$

Proof. Determinism ensures that for every product state $(s_{i,t}, s_{j,t})$ there is exactly one successor $(s_{i,t+1}, s_{j,t+1})$, determined by the mutual feedback of outputs. By induction on t, this yields a unique run of the product automaton starting from (s_i^0, s_j^0). Collecting the joint outputs at each step produces the word π, which is therefore unique. If the run stabilizes in a sink state with constant outputs, π is ultimately periodic otherwise π is infinite. □

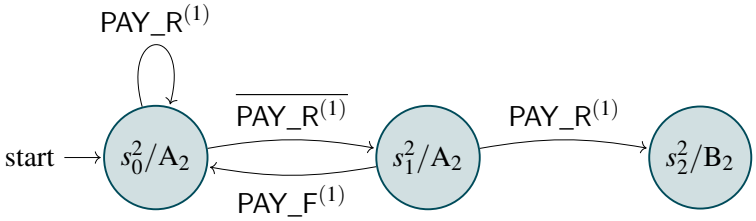
Example 48 (Product automaton sketch for $M_1^1 \times M_2^1$)**.** *We now project the two Moore machines M_1^1 (tenant) and M_2^1 (landlord) of Example 47 into their synchronous product $M_1^1 \times M_2^1$. Each state of the product records the joint output of both agents in that period, denoted by $\{A_1, A_2\}$, with $A_1 \in 2^{\Sigma_C^{(1)}}$ and $A_2 \in 2^{\Sigma_C^{(2)}}$. The initial state corresponds to (s_1^0, s_2^0); subsequent states reflect how both machines progress under synchronous feedback. Once a stable pair of states is reached, the product loops, generating the same joint output forever.*

The individual Moore machines underlying the product construction are recalled in Figure 4.3a and Figure 4.3b, while their synchronous product is summarized in Figure 4.3c and presented as a whole in Figure 4.3.

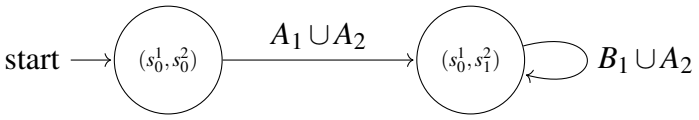
Concretely, The run of the product $M_1^1 \times M_2^1$ in Figure 4.3c, corresponds to the omega-word
$\langle \{\mathsf{PAY\_R}^{(1)}, \mathsf{OCC}^{(2)}, \mathsf{PAY\_R}^{(2)}\}\ (\{\mathsf{OCC}^{(1)}, \mathsf{PAY\_R}^{(1)}, \mathsf{OCC}^{(2)}, \mathsf{PAY\_R}^{(2)}\})^{\omega}\rangle$.



(a) M_1^1 (tenant), with
$A_1 = \{PAY\_R^{(1)}\}$, $B_1 = \{OCC^{(1)}, PAY\_R^{(1)}\}$, $C_1 = \emptyset$.



(b) M_2^1 (landlord), with : $A_2 = \{OCC^{(2)}, PAY\_R^{(2)}\}$, $B_2 = \emptyset$



(c) Product automaton $M_1^1 \times M_2^1$, with states as pairs (s_i^1, s_j^2) and transitions labeled by the joint outputs.

Figure 4.3: Tenant's machine M_1^1, landlord's machine M_2^1, and their product automaton $M_1^1 \times M_2^1$. Each product state is labeled by the joint output $\lambda_1(s_i^1) \cup \lambda_2(s_j^2)$. In the transition, the $\overline{PAY\_R^1}$ in the Moore machine M_j is a shorthand for any input action set not containing the action PAY_R^1, similarly PAY_R^1 is a shorthand for any input set containing PAY_R^1.

This section establishes a disciplined abstraction pipeline that bridges concrete executions and normative reasoning. Starting from raw metric-timed observations, we progressively lifted behavior into agent-local periodic traces and then into a collaboration-oriented representation aligned with the contractual calendar. The resulting model makes attempts, interference, and potential joint outcomes explicit without introducing interpretative assumptions or discarding observable behavior. Also, the modelization of strategies makes explicit how agents reason prospectively about each other's behavior, enabling the analysis of compliance not only as an ex post property of traces (after the interaction has occurred), but as an ex ante property of interacting plans (before actions are taken, at the level of intended behavior). Crucially, this abstraction separates what agents did from what jointly succeeded, preserving exactly the distinctions required for responsibility and blame attribution. The collaborative periodic trace thus serves as the semantic interface between concrete interaction data and the normative machinery introduced next.

4.4 The Two-Agents Collaborative Normative Logic TACNL

The abstraction developed in the previous section provides the execution model on which normative reasoning will be grounded. Having isolated agent-local attempts, interference, and collaborative outcomes at the level of contractual periods, we are now in a position to define a logic whose semantics are directly tied to this structure. The Two-Agents Collaborative Normative Logic (TACNL) is designed to speak about obligations, permissions, reparations, and powers over collaborative objectives, rather than low-level events. Its formulas are interpreted over the collaborative periodic traces constructed earlier, ensuring that normative evaluation is sensitive to attempts and obstruction rather than mere success or failure. In this way, TACNL does not abstract away interaction, but instead builds on the interaction model to provide precise, blame-aware contractual semantics.

4.4.1 Syntax of TACNL

In Figure 4.4, the syntax of TACNL is organized into three blocks: regular expressions, literals, and contracts. **Regular expressions (*re*).** This block specifies *when* clauses apply by describing patterns over monthly positions. An *atom* is a tagged action-set $A \subseteq \Sigma$ stating what the two parties did in a month. Complex expressions are formed with *union* $(re \mid re)$, *concatenation* $(re \cdot re)$ for next-month sequencing, *power* re^n (exactly n repetitions), and *Kleene plus* re^+ (one or more repetitions). The *wildcard* Γ is the union of all $A \subseteq \Sigma$ used to skip a position, and $\emptyset$ denotes the *empty set of actions*. These constructs enable patterns such as "a repair is requested this month" (an atom), "after any number of months if Agent 1 asks for termination $(\Gamma\cdot)$, or "three consecutive months" $(re)^3$.

Literals (ℓ). This block provides the primitive deontic statements for a single month: *obligation* $\mathbf{O}_p(a)$, *prohibition* $\mathbf{F}_p(a)$, and *power* $\mathbf{P}_p(a)$, plus the constants *valid* $\top$ and *invalid* $\bot$. Here $p \in \{1,2\}$ identifies the party (tenant or landlord) and $a \in \Sigma_C$ is a collaboration action. Intuitively, literals say *what* is required/forbidden/allowed of *whom*, independently of timing.

Contracts (C). This block composes literals into full specifications using: *conjunction* $(C \wedge C)$ to combine requirements in the same time position; *sequence* $(C;C)$ for next-time progression; *reparation* $(C \blacktriangleright C')$ for contrary-to-duty fall-backs saying you are requested to perform C, if you fail you must conform to C' in the next time position; *triggered* clauses $\langle\langle re \rangle\rangle C$ that activate C when a pattern re occurs; *guarded* clauses $\lceil re \rceil C$ that encapsulate conditions under which conforming to a contract is no longer necessary; *n-repetition* clauses enforces the repetition of the enforcement of a contract n times, with $n \in \mathbb{N}^*$; and finally, *repetition* $\mathbf{Rep}(C)$ for the infinite repetition of the enforcement of a

Figure 4.4: Syntax of TACNL

Given a collaboration alphabet Σ_C and a party index $p \in \{1,2\}$, we define the tagged action alphabet as $\Gamma := \Sigma_C^{(1)} \cup \Sigma_C^{(2)}$. Let $a \in \Sigma_C$ denote a collaborative action, $A \in 2^\Gamma$ a party-tagged action set, and $n \in \mathbb{N}^*$ a non-zero natural number. The syntax of TACNL is defined inductively by the grammar below:

Regular expressions	re	$::=$	A	[tagged action set]
		$\mid$	Γ	[wildcard]
		$\mid$	ε	[empty word]
		$\mid$	$\emptyset$	[empty set of actions]
		$\mid$	$(\mathsf{re} \mid \mathsf{re})$	[union]
		$\mid$	$\mathsf{re} \cdot \mathsf{re}$	[concatenation]
		$\mid$	re^n	[n-repetition]
		$\mid$	re^+	[Kleene plus]
Literals	ℓ	$::=$	$\mathbf{O}_p(a)$	[obligation]
		$\mid$	$\mathbf{F}_p(a)$	[prohibition]
		$\mid$	$\mathbf{P}_p(a)$	[permission]
		$\mid$	$\top$	[valid]
		$\mid$	$\bot$	[invalid]
Contracts	C	$::=$	ℓ	[literal]
		$\mid$	$C \wedge C$	[conjunction]
		$\mid$	$C;C$	[sequence]
		$\mid$	$C \blacktriangleright C$	[reparation]
		$\mid$	$\langle\langle re \rangle\rangle C$	[triggered]
		$\mid$	$\lceil re \rceil C$	[guarded]
		$\mid$	C^n	[n-repetition]
		$\mid$	$\mathbf{Rep}(C)$	[infinite-repetition]

contract. Together, these constructs could be used to capture the clauses of a contract, the conditions under which they are activated or terminated, and how the clauses relate to each other regarding reparations or the timing of their application. More specifically, the combination of repetition and a guarded contract can capture the notion of open-ended contracts. In the following example, we illustrate how we could capture our motivating example.

4.4.2 Illustrating the Encoding in TACNL for the Motivating Example

Example 49 (Encoding the rental clauses in TACNL)**.** *We now illustrate how the rental agreement introduced in Example 42 can be systematically encoded in the* TACNL *syntax (see Figure 4.4). We define the collaboration alphabet that captures all joint actions relevant to the contract:*

$$\Sigma_C = \{\mathsf{PAY\_R}, \mathsf{PAY\_F}, \mathsf{OCC}, \mathsf{Notif\_R}, \mathsf{Notif\_T}, \mathsf{Maint\_P}\}.$$

Each element corresponds to a collaborative outcome: PAY_R *(rent payment),* PAY_F *(late fee payment),* OCC *(occupancy),* Notif_R *(tenant's repair request),* Notif_T *(termination notice), and* Maint_P *(landlord performing repair).*

The encoding proceeds clause by clause, following the contract structure:

- ***C1 (Tenant pays rent):*** *The tenant is obliged to pay the rent each month.*

  $$C_1 := \mathbf{O}_1(\mathsf{PAY\_R}).$$

- ***C2 (Landlord guarantees occupancy):*** *The tenant gets the power to occupy the flat. Thus, the landlord is required not to interfere with the tenant's occupancy, encoded as a permission to allow the collaborative outcome* OCC*.*

  $$C_2 := \mathbf{P}_1(\mathsf{OCC}).$$

- ***C3 (Late-payment reparation):*** *Clause C3 introduces a contrary-to-duty (CTD) structure: if the tenant fails to fulfill the primary obligation (C1), a compensatory obligation to pay the late fee arises. This relationship is encoded as a reparation construct:*

  $$C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F}).$$

- ***C4 (Triggered repair request):*** *The tenant's request for repairs activates the landlord's duty to perform them within the following month. This is expressed using a triggered clause:*

$$C_4 := \langle\langle \{\mathsf{Notif\_R}^{(1)}\} \rangle\rangle \mathbf{O}_2(\mathsf{Maint\_P}).$$

- ***C5 (Termination and continuation):*** *The tenant may terminate the contract unilaterally by issuing a termination notice. Semantically, this action imposes a temporal limit on the otherwise open-ended obligations defined in the previous clauses. Specifically, once the termination notice is issued, the duties to guarantee occupancy (C_2), pay rent or reparations (C_3), and perform repairs (C_4) must persist for exactly three additional months before ceasing. To capture this behavior in* TACNL*, we do not define a single global guard; rather, we apply the guarding condition individually to each relevant sub-contract*[1]*. This ensures that each specific duty is correctly terminated after the notice period:*

$$C_{5\to 2} := \lceil \Gamma^+ \cdot \{\mathsf{Notif\_T}^{(1)}\} \cdot \Gamma^3 \rceil (C_2),$$
$$C_{5\to 3} := \lceil \Gamma^+ \cdot \{\mathsf{Notif\_T}^{(1)}\} \cdot \Gamma^3 \rceil (C_3),$$
$$C_{5\to 4} := \lceil \Gamma^+ \cdot \{\mathsf{Notif\_T}^{(1)}\} \cdot \Gamma^3 \rceil (C_4).$$

This step-by-step encoding shows how TACNL *integrates temporal regular patterns, deontic modalities, and event-triggered obligations in a single formalism. Clauses (C1–C5) together specify a full-cycle contract where collaborative actions such as payment, occupancy, repair, and termination are modeled as conditional and time-bounded obligations between the two agents.*

The syntax above fixes *what* can be expressed in TACNL. Regular-expression guards determine *when* a clause becomes relevant, while deontic literals determine *which* collaborative objective is required, forbidden, or allowed, and by which party. The encoding example shows that typical rental clauses can be represented compositionally using sequencing, reparation, triggering, and guarding.

Crucially, the interpretation of these collaborative objectives relies on the interaction model established in Section 4.3. Recall that we explicitly distinguished *attempts* captured in the periodic set traces from *successful collaboration* defined in Definition 34. Consequently, although the syntax refers to a collaborative action a, the evaluation logic is grounded in the agent-specific attempts $a^{(p)}$ preserved by our abstraction. This ensures that an agent is judged on their active participation rather than the joint outcome, preventing blame assignment when a compliant attempt is blocked by another party.

[1] The reason is that these clauses operate on distinct cycles; this concept will be clarified later in this section.

What remains is to define *how* such contracts are evaluated on traces. In TACNL, parts of a contract are *relative* to the prior satisfaction or violation of other parts of the same contract. In particular, reparations, guarded clauses, and termination conditions depend on identifying the exact prefix at which a primary obligation is first fulfilled or first breached. As a consequence, contract evaluation requires an exact and deterministic decomposition of traces into phases occurring before and after a decisive instant.

The next section introduces a non-classical, prefix-based semantic discipline that enforces such a decomposition. By isolating unique satisfaction and violation frontiers, this semantics ensures a single, well-defined progression of contractual states, which is essential for the correct interpretation of triggers, reparations, and contractual termination.

4.5 The Notion of Tight Semantics

In TACNL, contracts integrate responsibilities across multiple agents and temporal dimensions, making timing integral to their interpretation. Because every duty is anchored to a specific moment, failing to fulfill it punctually triggers an immediate contractual shift. Semantically, this necessitates partitioning a trace into a *pre-violation* prefix and a *post-violation* suffix, ensuring that reparation obligations are evaluated exclusively on the latter.

To ensure this evaluation is robust, the partitioning must be exact. We must identify a unique earliest violation point (or satisfaction point) to decompose observed behavior unambiguously. In the absence of such a unique boundary, reparations might be triggered prematurely, belatedly, or multiple times. To resolve this ambiguity, we introduce *tight forward* semantics:

- **Forward:** Verdicts are determined sequentially from left to right over prefixes, respecting the chronological order of the trace.
- **Tight:** The semantics anchor on the *first* decisive instant. This isolates a unique earliest satisfaction or violation frontier. Crucially, all strict extensions of this frontier are classified as "post-frontier," preventing the same responsibility from triggering multiple times.

The necessity of tightness. Without a mechanism to "lock in" a verdict at the earliest instance, significant semantic ambiguities arise. If we simply tag a prefix as "accepted" whenever it belongs to the target language L, we lose the distinction between the *moment* credit is earned and subsequent irrelevant behavior.

We illustrate this notion on regular languages. The Chomsky hierarchy[CS63] classifies formal languages according to their generative and computational complexity, ranging

from unrestricted languages down to regular ones. At the bottom of this hierarchy lie regular languages, which can be equivalently characterized by regular expressions, deterministic and nondeterministic finite automata, and monadic second-order logic over words. This equivalence is well understood and forms the backbone of automata-theoretic reasoning.

Consider the regular language $L = \{a\}$ ("seeing a once is success") over $\Sigma = \{a, b\}$.

- **Acceptance:** Reading a at the first position decides compliance immediately. Longer words like aa or ab must be treated as occurring *after* the decision, not as new acceptances. Without this distinction, we risk "double counting" compliance.
- **Rejection:** Conversely, reading b first establishes the earliest failure. Extensions like ba or bb are merely continuations of an already broken contract.

Our proposed five-valued semantics guarantees determinacy by identifying exactly one verdict per prefix: the earliest acceptance, the earliest rejection, post-acceptance, post-rejection, or pre-verdict (undecided).

4.5.1 From Regular Language Membership to Tight Prefix Languages

To generalize this construction to any regular language $L \subseteq \Sigma^*$, we must distinguish between classical *static language membership* and *behavioral evaluation*. While standard theory views a word w holistically (it is either in L or not), monitoring asks if the desired behavior has *already* been achieved.

Our objective is to partition Σ^* into regions that isolate these boundaries of decision. This requires explicitly resolving conflicts between "bad words" and "extensions of good words" in favor of the latter principle we term the *eager of acceptance and rejection*.

Topological Frontiers and Canonical Partition

We first identify candidate boundaries using standard topological operators.

Definition 39 (Closures and Frontiers)**.** *For a regular language $L \subseteq \Sigma^*$, we define:*

1. $\mathsf{PCl}(L)$*: The set of all prefixes of words in L*

$$\mathsf{PCl}(L) := \{u \in \Sigma^* \mid \exists v \in L : \ u \preceq v\}.$$

2. $\mathsf{BP}(L) = \Sigma^* \setminus \mathsf{PCl}(L)$: *The set of prefixes that cannot be extended to acceptance.*

$$\mathsf{BP}(L) := \{u \in \Sigma^* \mid u \notin \mathsf{PCl}(L)\}.$$

3. $\mathsf{Min}(L)$: *The set of the shortest elements from L.*

$$\mathsf{Min}(L) : \{u \in L \mid \forall u' \in:\ u \preceq u'\}.$$

Using $\mathsf{Min}(\cdot)$, we obtain candidates for eager acceptance ($\mathsf{Min}(L)$) and eager rejection ($\mathsf{Min}(\mathsf{BP}(L))$). However, overlaps may occur; a word can be a minimal bad prefix while extending a minimal accepted word (e.g., aa for $L = \{a\}$). To enforce the Priority of Acceptance, we define the canonical partition such that once acceptance is reached, extensions are permanently classified as irrelevant, even if they technically deviate from L.

Definition 40 (Canonical Semantic Partition)**.** *We partition Σ^* into five disjoint sets:*

$$\begin{aligned}
\mathsf{EA}(L) &:= \mathsf{Min}(L) && \textit{(Eager Acceptance)}\\
\mathsf{IAL}(L) &:= \mathsf{EA}(L)\,\Sigma^+ && \textit{(Irrelevant Acceptance)}\\
\mathsf{ER}(L) &:= \mathsf{Min}(\mathsf{BP}(L)) \setminus \mathsf{IAL}(L) && \textit{(Eager Rejection)}\\
\mathsf{IRL}(L) &:= \mathsf{ER}(L)\,\Sigma^+ && \textit{(Irrelevant Rejection)}\\
\mathsf{Pre}(L) &:= \mathsf{PCl}(\mathsf{EA}(L)) \setminus \mathsf{EA}(L) && \textit{(Pre-Verdict / Unknown)}
\end{aligned}$$

To formally encode the shift from membership to monitoring, the definition of $\mathsf{ER}(L)$ includes a subtraction operation. This ensures that a trace is only declared "bad" if it has not already been declared "good."

Properties of the Decomposition

We now verify that this construction correctly decomposes the language space. We use the notation $X = A \dot\cup B$ to denote that X is the union of disjoint sets A and B.

Lemma 28 (Structural Decomposition)**.** *For any $L \subseteq \Sigma^*$:*

1. $\mathsf{Min}(L)$ *and* $\mathsf{Min}(\mathsf{BP}(L))$ *are disjoint.*
2. *The closure splits into pre-verdict and acceptance regions:* $\mathsf{PCl}(\mathsf{Min}(L)) = \mathsf{Pre}(L) \dot\cup \mathsf{EA}(L)$.
3. *The total closure splits into minimal closure and extensions:* $\mathsf{PCl}(L) = \mathsf{PCl}(\mathsf{Min}(L)) \dot\cup \mathsf{IAL}(L)$.

4. *The bad class splits into eager and irrelevant rejection:* $\mathsf{BP}(L) = \mathsf{ER}(L) \dot\cup \mathsf{IRL}(L)$.

Proof. (1) follows because $\mathsf{Min}(L) \subseteq \mathsf{PCl}(L)$ and $\mathsf{Min}(\mathsf{BP}(L)) \subseteq \mathsf{BP}(L)$, which are complements. (2) is immediate by definition. For (3), take $u \in \mathsf{PCl}(L)$. There exists $z \in L$ such that $u \preceq z$. Let m be the shortest accepted prefix of z ($m \in \mathsf{Min}(L)$). Either $u \preceq m$ (so $u \in \mathsf{PCl}(\mathsf{Min}(L))$) or $m \prec u$ (so $u \in m\Sigma^+ \subseteq \mathsf{IAL}(L)$). For (4), any $u \in \mathsf{BP}(L)$ has a unique shortest bad prefix $b \in \mathsf{Min}(\mathsf{BP}(L))$. If $b \in \mathsf{IAL}(L)$, then u is also in $\mathsf{IAL}(L)$ (contradicting $u \in \mathsf{BP}(L)$). Thus $b \notin \mathsf{IAL}(L)$, implying $b \in \mathsf{ER}(L)$. Therefore, u is either b itself or an extension, covering $\mathsf{ER}(L) \cup \mathsf{IRL}(L)$. □

Lemma 29 (Cross Disjointness)**.** *The sets* $\mathsf{PCl}(L)$, $\mathsf{ER}(L)$, $\mathsf{IRL}(L)$, *and* $\mathsf{IAL}(L)$ *are mutually disjoint where required:*

$$\mathsf{PCl}(L) \cap \mathsf{BP}(L) = \emptyset, \qquad \mathsf{IRL}(L) \cap \mathsf{IAL}(L) = \emptyset, \qquad \mathsf{Min}(L) \cap \mathsf{IRL}(L) = \emptyset.$$

Proof. The first is by definition. For the second, assume $u \in \mathsf{IRL}(L) \cap \mathsf{IAL}(L)$. Then u extends both a minimal bad prefix $b \in \mathsf{ER}(L)$ and a minimal accepted word $m \in \mathsf{EA}(L)$. If $m \preceq b$, then b extends an accepted word, so $b \in \mathsf{IAL}(L)$, contradicting $b \in \mathsf{ER}(L)$. If $b \preceq m$, then m extends a bad prefix, implying $m \in \mathsf{BP}(L)$, contradicting $m \in L$. Finally, $\mathsf{Min}(L) \cap \mathsf{IRL}(L) = \emptyset$ because $\mathsf{Min}(L) \subseteq \mathsf{PCl}(L)$ while $\mathsf{IRL}(L) \subseteq \mathsf{BP}(L)$. □

Theorem 7 (Five-way partition of Σ^*)**.** *For every* $L \subseteq \Sigma^*$*, the space of all possible words is decomposed into:*

$$\Sigma^* = \mathsf{Pre}(L) \dot\cup \mathsf{EA}(L) \dot\cup \mathsf{ER}(L) \dot\cup \mathsf{IAL}(L) \dot\cup \mathsf{IRL}(L).$$

Proof. Combining Lemma 28 and Lemma 29, we have $\Sigma^* = \mathsf{PCl}(L) \dot\cup \mathsf{BP}(L)$. Substituting the decompositions for $\mathsf{PCl}(L)$ and $\mathsf{BP}(L)$ yields the result. □

Tight Five-Valued Semantics

We move now to define the prefix-level semantics $[\![u \vDash L]\!]_5$ using the values $\mathbb{V}_5 = \{?, \top^t, \bot^t, \top^p, \bot^p\}$. These correspond to pre-verdict, eager acceptance, eager rejection, irrelevant acceptance, and irrelevant rejection, respectively.

Definition 41 (Five-valued prefix semantics)**.** *Fix* $L \subseteq \Sigma^*$*. For any* $u \in \Sigma^*$*:*

$$[\![u \vDash L]\!]_5 := \begin{cases} ? & \text{if } u \in \mathsf{Pre}(L) \\ \top^t & \text{if } u \in \mathsf{EA}(L) \\ \bot^t & \text{if } u \in \mathsf{ER}(L) \\ \top^p & \text{if } u \in \mathsf{IAL}(L) \\ \bot^p & \text{if } u \in \mathsf{IRL}(L) \end{cases}$$

This semantics guarantees key stability properties essential for contract monitoring:

Theorem 8 (Prefix Monotonicity and Determinacy)**.** *For any $u \in \Sigma^*$:*

1. ***Determinism:*** *$[\![u \vDash L]\!]_5$ yields exactly one verdict (by Theorem 7).*
2. ***Monotone evolution:*** *If $[\![u \vDash L]\!]_5 \in \{\top^t, \top^p\}$, then for all $x \in \Sigma^+$, $[\![u \circ x \vDash L]\!]_5 = \top^p$. Symmetrically, if $[\![u \vDash L]\!]_5 \in \{\bot^t, \bot^p\}$, then for all $x \in \Sigma^+$, $[\![u \circ x \vDash L]\!]_5 = \bot^p$.*
3. ***Unique decision frontier:*** *If $[\![u \vDash L]\!]_5 = \top^p$, there exists a unique strict prefix $u' \prec u$ such that $[\![u' \vDash L]\!]_5 = \top^t$. The same uniqueness holds for rejection.*

Proof. **Monotonicity:** If $[\![u \vDash L]\!]_5 = \top^t$, then $u \in \mathsf{EA}(L)$. Any extension $u \circ x$ belongs to $\mathsf{EA}(L)\,\Sigma^+ = \mathsf{IAL}(L)$, yielding $\top^p$. If $u \in \mathsf{IAL}(L)$, it is already an extension of a word in $\mathsf{EA}(L)$, so further extensions remain in $\mathsf{IAL}(L)$. The rejection case follows identical logic using $\mathsf{ER}(L)$ and $\mathsf{IRL}(L)$.

Unique Frontier: If $u \in \mathsf{IAL}(L)$, then $u = m \circ x$ for some $m \in \mathsf{EA}(L)$. Since $\mathsf{EA}(L)$ contains *minimal* accepted words, no strict prefix of m is in L, and no strict extension of m (that is a prefix of u) can be in $\mathsf{EA}(L)$. Thus, m is the unique frontier point. □

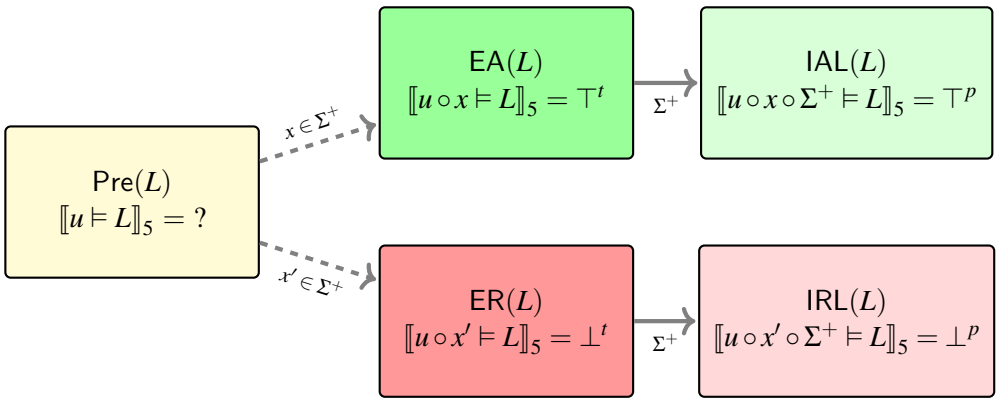


Figure 4.5: Visualizing the five-way semantic partition (Theorem 8). The diagram illustrates the irreversible flow of a trace from the undecided state (Yellow) to a decisive frontier (Eager Acceptance in Green or Eager Rejection in Red) and subsequently to an irrelevant post-verdict state. Arrows indicate possible transitions via trace extension; dashed arrows stands for the possible existence of a suffix; plain arrows stands for any non-empty suffix leads to the next verdict; missing arrows imply that no direct transition is possible.

Example 50 (Five-region decomposition and Trace Evolution)**.** *Let $\Sigma = \{a, b\}$ and let $L = \{a\}$. The goal is to detect the occurrence of the symbol a immediately.*

1. Static Partitioning. *First, we identify the topological frontiers.*

$$\mathsf{PCI}(L) = \{\varepsilon, a\},$$
$$\mathsf{BP}(L) = \Sigma^* \setminus \{\varepsilon, a\} = \{b, aa, ab, \dots\}.$$

The candidates for eager frontiers are $\mathsf{Min}(L) = \{a\}$ *and* $\mathsf{Min}(\mathsf{BP}(L)) = \{b, aa\}$.

Crucially, aa is a bad prefix in the static sense (it is not in L), but it extends the accepted word a. To preserve tight determinacy, the acceptance of a takes priority. We formally compute the partition sets:

- $\mathsf{EA}(L) = \{a\}$ *(the first success).*
- $\mathsf{ER}(L) = \{b, aa\} \setminus (\{a\}\Sigma^+) = \{b\}$ *(the first failure). Note that aa is removed from the rejection set.*
- $\mathsf{IAL}(L) = \{aa, ab, aaa, \dots\}$ *(extensions of success).*

2. Trace Evolution. *We observe how the semantics* $[\![u \vDash L]\!]_5$ *evolve step-by-step as we process two different input streams.*

Trace A (Success): *The user inputs a, then continues with b.*

$$\underbrace{\varepsilon}_{?\,(\in \mathsf{Pre}(L))} \xrightarrow{a} \underbrace{a}_{\top^t\,(\in \mathsf{EA}(L))} \xrightarrow{b} \underbrace{ab}_{\top^p\,(\in \mathsf{IAL}(L))} \xrightarrow{\forall u \in \Sigma^*} \top^p$$

Observation: *At* $u = a$*, the verdict is locked to* $\top^t$*. The subsequent b does not trigger a violation; it is merely an irrelevant extension (*$\top^p$*).*

Trace B (Failure): *The user inputs b, then continues with a.*

$$\underbrace{\varepsilon}_{?\,(\in \mathsf{Pre}(L))} \xrightarrow{b} \underbrace{b}_{\bot^t\,(\in \mathsf{ER}(L))} \xrightarrow{a} \underbrace{ba}_{\bot^p\,(\in \mathsf{IRL}(L))} \xrightarrow{\forall u \in \Sigma^*} \bot^p$$

Observation: *At* $u = b$*, the verdict is locked to* $\bot^t$*. Even though the next symbol is an a (which usually signifies success), it arrives too late. The trace is already strictly post-violation (*$\bot^p$*), ensuring the contract is not "fulfilled" after it has already been broken.*

Having established the theoretical partitions of for any regular language L, we move now operationalize them by constructing a Moore machine monitor from a deterministic finite automaton recognizing L.

From Language Automaton to Tight Monitor Construction

To implement the tight semantics operationally, we now lift the standard deterministic language automaton into a tight monitor. While a standard automaton recognizes complete words holistically, a tight monitor must instead classify every individual prefix into one of the five semantic regions. We achieve this by preserving the underlying transition structure of the original automaton while augmenting its output behavior: every state is assigned a tight verdict corresponding to the five-valued prefix semantics. The result is the tight monitor formalized below.

We first define the auxiliary components required for this transformation before combining them into the final Moore machine.

Definition 42 (Five-region automata)**.** *Let $L \subseteq \Sigma^*$ be a regular language and let*

$$\mathcal{A}(L) = (Q, \Sigma, \delta, q_0, F) \quad \textit{with} \quad \mathcal{L}(\mathcal{A}(L)) = L$$

be a deterministic finite automaton (DFA) for L. We define the following derived DFAs, all over the same alphabet Σ:

- $\mathcal{A}_{\mathrm{EA}}(L) := (Q_{\mathrm{EA}}, \Sigma, \delta_{\mathrm{EA}}, q^0_{\mathrm{EA}}, F_{\mathrm{EA}})$ *with* $\mathcal{L}(\mathcal{A}_{\mathrm{EA}}(L)) = \mathsf{EA}(L)$.
- $\mathcal{A}_{\mathrm{PRE}}(L) := (Q_{\mathrm{PRE}}, \Sigma, \delta_{\mathrm{PRE}}, q^0_{\mathrm{PRE}}, F_{\mathrm{PRE}})$ *with* $\mathcal{L}(\mathcal{A}_{\mathrm{PRE}}(L)) = \mathsf{Pre}(L)$.
- $\mathcal{A}_{\mathrm{IAL}}(L) := (Q_{\mathrm{IAL}}, \Sigma, \delta_{\mathrm{IAL}}, q^0_{\mathrm{IAL}}, F_{\mathrm{IAL}})$ *with* $\mathcal{L}(\mathcal{A}_{\mathrm{IAL}}(L)) = \mathsf{IAL}(L)$.
- $\mathcal{A}_{\mathrm{ER}}(L) := (Q_{\mathrm{ER}}, \Sigma, \delta_{\mathrm{ER}}, q^0_{\mathrm{ER}}, F_{\mathrm{ER}})$ *with* $\mathcal{L}(\mathcal{A}_{\mathrm{ER}}(L)) = \mathsf{ER}(L)$.
- $\mathcal{A}_{\mathrm{IRL}}(L) := (Q_{\mathrm{IRL}}, \Sigma, \delta_{\mathrm{IRL}}, q^0_{\mathrm{IRL}}, F_{\mathrm{IRL}})$ *with* $\mathcal{L}(\mathcal{A}_{\mathrm{IRL}}(L)) = \mathsf{IRL}(L)$.

These automata are obtained using standard automata-theoretic constructions, including prefix-closure, complement, breadth-first identification of minimal accepting prefixes, and right-ideal saturation, as commonly used in DFA analysis [HMU01].

Definition 43 (Five-region Moore machine)**.** *Let $L \subseteq \Sigma^*$ be a regular language and let the five DFAs from Definition 42 be given, with accepting sets $F_{\mathrm{PRE}}, F_{\mathrm{EA}}, F_{\mathrm{ER}}, F_{\mathrm{IAL}}, F_{\mathrm{IRL}}$. The* five-region Moore machine *for L is defined as:*

$$\mathcal{M}_{5tight}(L) \;:=\; \big(S,\; s^0,\; \Sigma,\; \mathbb{V}_5,\; \delta,\; \lambda\big),$$

where the state space is the Cartesian product

$$S := Q_{\mathrm{PRE}} \times Q_{\mathrm{EA}} \times Q_{\mathrm{ER}} \times Q_{\mathrm{IAL}} \times Q_{\mathrm{IRL}},$$

the initial state is the tuple of initial states

$$s^0 := \big(q^0_{\mathrm{PRE}},\, q^0_{\mathrm{EA}},\, q^0_{\mathrm{ER}},\, q^0_{\mathrm{IAL}},\, q^0_{\mathrm{IRL}}\big),$$

the transition function operates component-wise

$$\delta\big((p,e,r,a,\rho),\sigma\big) := \big(\delta_{\mathrm{PRE}}(p,\sigma),\ \delta_{\mathrm{EA}}(e,\sigma),\ \delta_{\mathrm{ER}}(r,\sigma),\ \delta_{\mathrm{IAL}}(a,\sigma),\ \delta_{\mathrm{IRL}}(\rho,\sigma)\big),$$

and the output function λ assigns verdicts based on component acceptance

$$\lambda(p,e,r,a,\rho) := \begin{cases} \top^t & \text{if } e \in F_{\mathrm{EA}}, \\ \bot^t & \text{if } r \in F_{\mathrm{ER}}, \\ \top^p & \text{if } a \in F_{\mathrm{IAL}}, \\ \bot^p & \text{if } \rho \in F_{\mathrm{IRL}}, \\ ? & \text{if } p \in F_{\mathrm{PRE}}. \end{cases}$$

Since L uniquely determines all components, we write $\mathcal{M}_{5tight}(L)$ as a direct function of the language.

Example 51 (Compact five-valued Moore monitor)**.** *Consider the language $L = \{a, ab, bb\}$ over $\Sigma = \{a, b\}$. The minimized five-output Moore machine determines verdicts by tracking the unique accepting component among the regions* $\mathsf{Pre}(L), \mathsf{EA}(L), \mathsf{ER}(L), \mathsf{IAL}(L), \mathsf{IRL}(L)$.

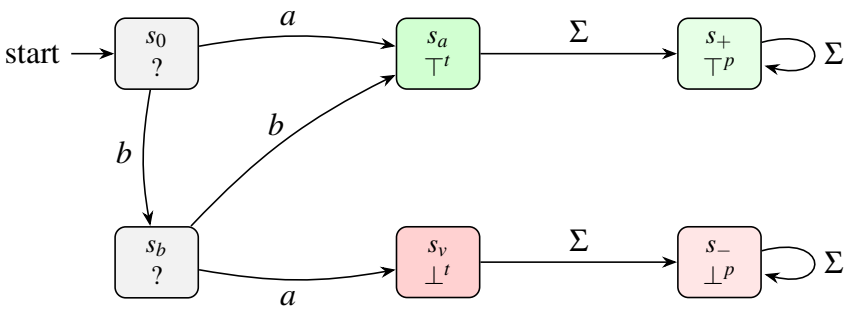


Figure 4.6: Minimized five-valued Moore machine for $L = \{a, ab, bb\}$. Each node displays its identifier and output verdict. Green states indicate satisfaction, red states indicate violation, and gray states remain undecided. Starting from s_0, inputs a or bb yield $\top^t$, input ba yields $\bot^t$, and subsequent inputs transition to the irreversible post-frontier states $\top^p$ or $\bot^p$.

Lemma 30 (Correctness of the tight-product Moore machine)**.** *Let $L \subseteq \Sigma^*$ be regular and let $M_{5tight}(L)$ be the machine from Definition 43. For every input prefix $u \in \Sigma^*$, the machine output coincides with the semantic definition:*

$$[\![u \vDash L]\!]_5 \ = \ \lambda\big(\delta^*(s^0, u)\big).$$

Proof. By construction, the global state reached after reading u is the tuple of states reached by the five component DFAs.

$$\delta^*(s^0,u) = \big(\delta^*_{\mathrm{PRE}}(q^0_{\mathrm{PRE}},u), \ldots, \delta^*_{\mathrm{IRL}}(q^0_{\mathrm{IRL}},u)\big).$$

Since each component DFA recognizes exactly one of the five semantic regions (e.g., $u \in \mathsf{EA}(L) \iff \delta^*_{\mathrm{EA}}(\dots) \in F_{\mathrm{EA}}$), and since Theorem 7 guarantees these regions form a disjoint partition, exactly one component will accept. The output function λ then maps this unique acceptance to the corresponding verdict, matching Definition 41. □

Theorem 9 (Linear Complexity of Tight Monitors)**.** *Let $\mathcal{A}(L) = (Q, \Sigma, \delta, q_0, F)$ be a complete DFA for L, and let $\mathcal{M}_{5tight}(L)$ be the tight five-valued Moore machine constructed in Definition 43. The number of reachable states in the tight monitor is bounded linearly by the size of the input DFA:*

$$\left|S_{\mathrm{reach}}(\mathcal{M}_{5tight}(L))\right| \leq |Q|.$$

Proof. The validity of this bound rests on the structural invariance of the component automata.

Structural Invariance. To construct the five region automata ($\mathcal{A}_{\mathrm{PRE}}, \mathcal{A}_{\mathrm{EA}}$, etc.), we modify *only* the accepting sets (F) of the original complete DFA $\mathcal{A}(L)$. The transition structure (Q, Σ, δ, q_0) is preserved identically across all five components. We explicitly avoid minimization techniques on the individual components that would prune states or merge sinks, as this would break the structural alignment.

The Diagonal Argument. The state space of the Moore machine is formally the Cartesian product $S = Q \times Q \times Q \times Q \times Q$. However, since the transition function δ is identical for every component, for any input prefix $u \in \Sigma^*$, the state reached by every component is the same:

$$\delta^*_{\mathrm{PRE}}(q_0, u) = \cdots = \delta^*_{\mathrm{IRL}}(q_0, u) = \delta^*(q_0, u).$$

Consequently, the set of reachable states S_{reach} is restricted strictly to the diagonal of the product space:

$$S_{\mathrm{reach}} = \{(q, q, q, q, q) \mid q \in \mathrm{Reach}(\mathcal{A}(L))\}.$$

Because there exists a bijection between the reachable states of $\mathcal{A}(L)$ and $\mathcal{M}_{5\mathrm{tight}}(L)$, the linear bound holds. □

Summary This section has developed a *correct-by-construction* framework that transforms any regular language $L \subseteq \Sigma^*$ first into the five semantic regions required by tight monitoring, and subsequently into a single executable monitor. This workflow relies exclusively on standard automata-theoretic operations—completion, complement, breadth-first search for minimal frontiers, and right-ideal saturation—applied to the underlying DFA graph.

Conceptually, this construction bridges the gap between the linguistic requirements of normative systems and operational verification. It ensures that norms enter into force at exact positions and that monitor decisions are both fair (no premature verdicts) and final (no evolution to the opposite verdict). By strictly preserving the transition structure, we deliver a monitor that is correct by design without incurring the state explosion typical of product constructions.

4.5.2 Illustration Through Regular Expressions from TACNL

Having defined the general construction, we now illustrate it on TACNL regular expressions. The procedure begins by specifying the language semantics of the regular expression, followed by a standard transformation into a deterministic automaton, and finally lifting it into the five-valued monitor. This subsection fixes the semantics of the regular expressions that act as temporal guards in TACNL.

Semantics for Regular Expressions from TACNL

We fix Σ_C as the collaboration alphabet and $\Sigma_C^{(p)}$, the tagged collaboration alphabet of agent $p \in \{1,2\}$, and the *letter alphabet* $\Gamma := 2^{\Sigma_C^{(1)} \cup \Sigma_C^{(2)}}$ from the syntax in Figure Figure 4.4, ranging over all possible sets of agent indexed actions that may occur jointly within a single contractual period.

Definition 44 (Semantics of regular expressions)**.** *The satisfaction relation for a regular expression* re*, written* $\pi \models_{\mathsf{re}} \mathsf{re}$*, is defined over a finite trace* $\pi = \langle A_1, \ldots, A_m \rangle \in \Gamma^*$*, where each letter* $A_i \in \Gamma$ *denotes the set of actions that occurred at period i. The relation is defined inductively as follows:*

$$
\begin{aligned}
&\pi \models_{\mathsf{re}} \mathsf{A} && \text{iff} \quad |\pi| = 1 \text{ and } \mathsf{A} \subseteq A_1,\\
&\pi \models_{\mathsf{re}} \Gamma && \text{iff} \quad |\pi| = 1,\\
&\langle - \rangle \models_{\mathsf{re}} \varepsilon,\\
&\pi \models_{\mathsf{re}} \emptyset && \text{iff} \quad |\pi| = 1 \text{ and } A_1 = \emptyset,\\
&\pi \models_{\mathsf{re}} \mathsf{re}_1 \mid \mathsf{re}_2 && \text{iff} \quad (\pi \models_{\mathsf{re}} \mathsf{re}_1) \text{ or } (\pi \models_{\mathsf{re}} \mathsf{re}_2),\\
&\pi \models_{\mathsf{re}} \mathsf{re}_1 \cdot \mathsf{re}_2 && \text{iff} \quad \exists k \in \{0,\ldots,m\} : \pi_k \models_{\mathsf{re}} \mathsf{re}_1 \text{ and } \pi^k \models_{\mathsf{re}} \mathsf{re}_2,\\
&\pi \models_{\mathsf{re}} \mathsf{re}^1 && \text{iff} \quad \pi \models_{\mathsf{re}} \mathsf{re},\\
&\pi \models_{\mathsf{re}} \mathsf{re}^n && \text{iff} \quad \pi \models_{\mathsf{re}} \mathsf{re} \cdot \mathsf{re}^{n-1} \quad \text{for } n > 1,\\
&\pi \models_{\mathsf{re}} \mathsf{re}^+ && \text{iff} \quad \exists n \geq 1 : \pi \models_{\mathsf{re}} \mathsf{re}^n.
\end{aligned}
$$

We write $\mathcal{L}(\mathsf{re}) := \{\pi \in \Gamma^* \mid \pi \models_{\mathsf{re}} \mathsf{re}\}$ *for the language denoted by* re.

Reading the clauses.

- **Atom** A**.** Matches exactly one period: $\pi = \langle A_0 \rangle$ with $\mathsf{A} \subseteq A_0$.
- **Wildcard** Γ**.** Matches any single period: $\pi = \langle A_0 \rangle$ for arbitrary $A_0 \in \Gamma$.
- **Empty word** ε**.** Matches only the empty word: $\pi = \langle - \rangle$.
- **Empty-action letter** $\emptyset$**.** Matches the one-period trace with no actions: $\pi = \langle \emptyset \rangle$.
- **Union** $(re_1 \mid re_2)$**.** Holds iff at least one disjunct holds on the whole trace.
- **Sequencing** $(\mathsf{re}_1 \cdot \mathsf{re}_2)$**.** There exists a split index $k \in \{0, \ldots, n\}$ such that the prefix $\pi[1,k]$ satisfies re_1 and the suffix $\pi[k+1,n]$ satisfies re_2. The case $k = 0$ corresponds to re_1 matching the empty trace, while $k = n$ corresponds to re_2 matching the empty trace. Thus, sequencing expresses that the trace can be decomposed into two contiguous segments, evaluated independently and in order.
- **Fixed power** re^n**.** Iterated sequencing of re exactly n times is inductively defined: if $n = 1$ then evaluate for re; for $n > 1$, enroll the first occurrence of re and concatenate with re^{n-1}.
- **Kleene plus** re^+**.** Some positive iteration holds: $\exists n \geq 1$ with $\pi \models_{\mathsf{re}} re^n$.

Automata Construction Pipeline

We now give a concrete, standard pipeline that realizes the semantics of Definition 44 *exactly* by an automaton over the alphabet $\Gamma = 2^{\Sigma_C^{(1)} \cup \Sigma_C^{(2)}}$.

We build $\mathcal{A}_\varepsilon(re)$ by structural recursion on re, using the usual two distinguished states $(s_{\text{in}}, s_{\text{out}})$ per fragment and ε-transitions for wiring [Tho68, HMU01, Sip12]. The only twist is how we treat letters, since an atom A matches *any* Γ-letter X that *covers* A (Definition 44).

- **Atom** $\mathsf{A} \subseteq \Gamma$**:** create two states $p \to q$ and add, for *every* $X \in \Gamma$ with $\mathsf{A} \subseteq X$, a transition $p \xrightarrow{X} q$. This enforces "one period, with all actions in A present."
- **Wildcard** Γ**:** create $p \xrightarrow{X} q$ for *all* $X \in \Gamma$.
- **Empty word** ε**:** create $p \xrightarrow{\varepsilon} q$.
- **Empty-action letter** $\emptyset$**:** create a single-letter fragment $p \xrightarrow{\{\emptyset\}} q$ (i.e., only the Γ-letter $\emptyset$).

- **Union** $(re_1 \mid re_2)$**:** build fragments for re_1 and re_2 with entries/exits (p_1, q_1) and (p_2, q_2). Add fresh p, q and wire $p \xrightarrow{\varepsilon} p_1$, $p \xrightarrow{\varepsilon} p_2$, $q_1 \xrightarrow{\varepsilon} q$, $q_2 \xrightarrow{\varepsilon} q$.
- **Sequencing** $(re_1 \cdot re_2)$**:** build (p_1, q_1) and (p_2, q_2), then add $q_1 \xrightarrow{\varepsilon} p_2$ and take (p_1, q_2) as entry/exit. This matches the semantic split $\pi[1,k]$ and $\pi[k+1,n]$, with $k = 0$ denoting the empty prefix.
- **Fixed power** re^n**:** unroll as $re; \cdots; re$ (n times). The base $re^1 \equiv re$.
- **Kleene plus** re^+**:** build (p_1, q_1) for re, then add $q_1 \xrightarrow{\varepsilon} p_1$ and take (p_1, q_1) as entry/exit. (At least one iteration is enforced by entering at p_1.)

The resulting NFA accepts exactly $\mathcal{L}(re)$.

Stage 2: Determinization. Apply the standard subset construction with ε-closures to obtain a DFA $\mathcal{A}_D(re) = (Q, \Gamma, \delta, q_0, F)$ that recognizes the same language [RS59, HMU01].

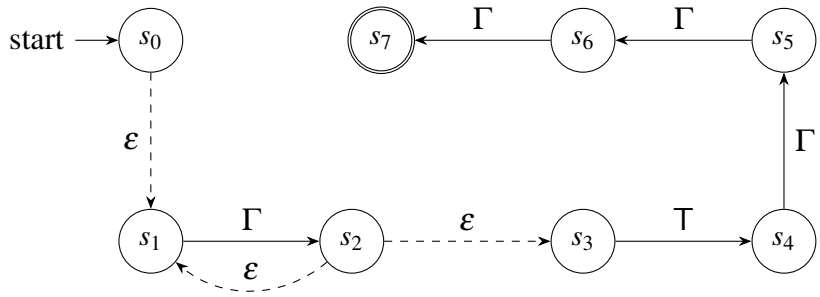
Stage 3: Completion, trimming, and minimization. For monitoring we assume a *completed* DFA: add (if needed) a sink state so that δ is total on $Q \times \Gamma$. Next, trim unreachable states (retain only $\mathsf{Reach}(\mathcal{A}_D(re))$). Optionally, apply standard DFA minimization (for instance via partition refinement, as in Hopcroft's algorithm) to obtain an equivalent smallest DFA up to isomorphism. This step is semantics-preserving and is used only to reduce the state space before the five-region constructions and the Moore lifting.

Correctness Sketch. By structural induction on re. The letter fragments implement exactly the one-step clauses (A, $*$, $\emptyset$), union and sequencing are the usual ε-wiring proofs, and re^n (unrolled) and re^+ (loop back from the exit) match the inductive clauses in Definition 44. Determinization and completion preserve language.

Example 52 (End-to-end construction)**.** *Continuing from Example 49, we demonstrate the automata construction for* C_5*:*

$$\begin{aligned} \mathsf{re}_{C_5} &:= \Gamma^+ \cdot \{\mathsf{Notif\_T}^{(1)}\} \cdot \Gamma^3, \\ \Gamma &:= 2^{\Sigma_C^{(1)} \cup \Sigma_C^{(2)}}, \\ \Sigma_C &:= \{\mathsf{PAY\_R},\ \mathsf{PAY\_F},\ \mathsf{OCC},\ \mathsf{Notif\_R},\ \mathsf{Notif\_T},\ \mathsf{Maint\_P}\}. \end{aligned}$$

Figure Figure 4.7 depicts the Thompson-style ε*-NFA for* re_{C_5}*, and Figure Figure 4.8 shows its determinized and completed DFA.*



Γ: any non-empty set of actions T: set A with $\mathsf{Notif\_T}^{(1)} \in A$ dashed ε: wiring

Figure 4.7: Thompson-style ε-NFA for $\mathsf{re}_{C_5} = \Gamma^+ \cdot \{\mathsf{Notif\_T}^{(1)}\} \cdot \Gamma^3$ over $\Gamma = 2^\Sigma$. From s_0 we enter the Γ^+ block ($s_1 \xrightarrow{\Gamma} s_2$ with a back ε-loop to enforce "one or more" steps), then take a single T-labeled letter (the period that contains $\mathsf{Notif\_T}^{(1)}$), followed by exactly three arbitrary periods (three Γ transitions) to the accepting state s_7. Determinization and completion of this NFA yield a DFA that recognizes precisely the denotation of re_{C_5} in Definition 44.

We now close the pipeline by instantiating the tight monitor construction. The construction below wraps the language DFA for $\mathcal{L}(\mathsf{re})$ into the five-region Moore machine, yielding a monitor whose output coincides with the five-valued prefix semantics on all prefixes.

Definition 45 (Tight monitor construction for regular expressions)**.** *Let* re *be a regular expression over the alphabet* Γ *and let* $L := \mathcal{L}(\mathsf{re}) \subseteq \Gamma^*$ *be its language as in Definition 44. Let*

$$\mathcal{M}_{5tight}(L) \;=\; \big(S, s^0, \Gamma, \mathbb{V}_5, \delta, \lambda\big)$$

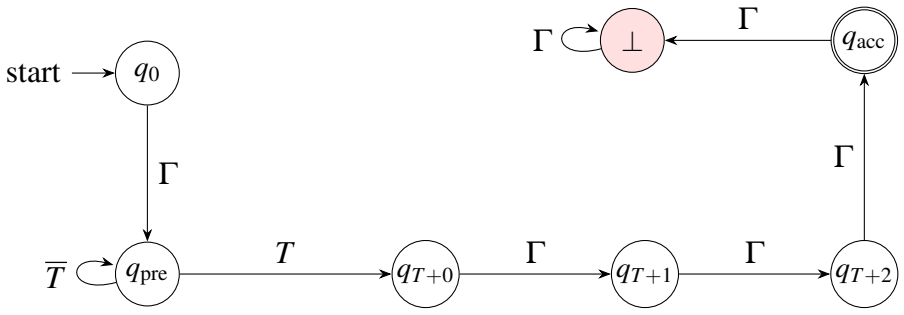


Figure 4.8: Determinized and completed DFA for $\mathsf{re}_{C_5} = \Gamma^+ \cdot \{\mathsf{Notif\_T}^{(1)}\} \cdot \Gamma^3$ over the alphabet $\Gamma = 2^\Sigma$. State q_{pre} collects the initial Γ^+ segment; transition on T begins the "+3 letters" counter ($q_{T+0} \to q_{T+1} \to q_{T+2} \to q_{\text{acc}}$). Any overrun moves to the sink.

be the five-region Moore machine for L from Definition 43. The tight satisfaction monitor *for* re *is this Moore machine:*

$$\mathsf{TSMC}_{\mathsf{re}}(\mathsf{re}) := \mathcal{M}_{5tight}(\mathcal{L}(\mathsf{re})).$$

Example 53 (Tight monitor for C_5)**.** *Continuing Example 52, Figure Figure 4.9 shows the compact five-valued Moore monitor obtained by applying the tight monitor construction of Definition 45 to the regular expression* re_{C_5}.

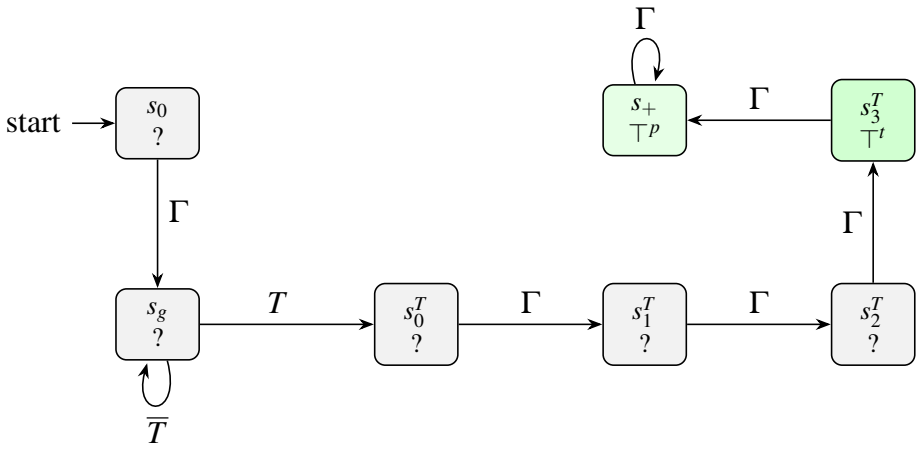


Figure 4.9: Compact five-valued Moore monitor for $re_{C_5} = \Gamma^+ \cdot \{\mathsf{Notif\_T}^{(1)}\} \cdot \Gamma^3$.

4.6 Tight Forward Reasoning on Contract Compliance

Methodological overview. This section develops the forward-looking semantics and monitoring constructions in a layered manner. We first introduce tight satisfaction and violation as *frontier-based* judgements over prefixes of a synchronous periodic trace. These judgements identify the unique decisive point at which a contract becomes irrevocably satisfied or violated. On top of this core semantics, we derive coarser verdicts, prove coherence properties, and construct Moore-style monitors that operationalize the semantics. The continuation of this development extends the same methodology to responsibility-aware verdicts and blame monitoring.

4.6.1 Denotational Semantics for Forward-Looking Tight Contract Satisfaction

Fix the tagged collaboration alphabet $\Sigma = \Sigma_C^{(1)} \cup \Sigma_C^{(2)}$ and the induced letter alphabet $\Gamma = 2^{\Sigma}$. Traces, prefixes, suffixes, and their basic operators are as defined in Section Section 2.2. In this subsection, we only introduce the forward-looking semantic judgements used to evaluate contracts along such traces.

Core tight judgements. All semantic clauses below are stated relative to the prefix structure fixed in Section Section 2.2.

We define two tight relations (inductively on the syntax of C):

$$\pi \models_{\top^t} C \quad \text{(tight satisfaction)}, \qquad \pi \models_{\bot^t} C \quad \text{(tight violation)}.$$

Intuitively, $\models_{\top^t}$ holds exactly at the *first prefix* where the contract becomes satisfied (acceptance frontier), and $\models_{\bot^t}$ holds exactly at the *first prefix* where it becomes violated (rejection frontier).

Derived judgements. Using these frontiers, we define the remaining derived relations of the five-valued semantics. By Lemma 34, at most one tight frontier can occur along a fixed trace, so the clauses below classify a prefix by its position relative to this (unique, if it exists) decisive point.

Definition 46 (Post and Pre-satisfaction semantic definition)**.** *For a contract C and trace π the Pre satisfaction relation $\models_?$, the post satisfaction and violation relations, respectively $\models_{\top^p}$ and $\models_{\bot^p}$ are defined on the structure of the trace π and the tight satisfaction and violation relations:*

$$\begin{aligned}
\pi \models_? C &\iff \forall k < |\pi| : \neg\big(\pi[1,k] \models_{\top^t} C\big) \text{ and } \neg\big(\pi[1,k] \models_{\bot^t} C\big), \\
\pi \models_{\top^p} C &\iff \exists k < |\pi| : \pi[1,k] \models_{\top^t} C, \\
\pi \models_{\bot^p} C &\iff \exists k < |\pi| : \pi[1,k] \models_{\bot^t} C.
\end{aligned}$$

Each trace prefix is classified relative to the earliest decisive prefix: before it the trace is undecided ($\models_?$), at it the contract is decided tightly ($\models_{\top^t}$ or $\models_{\bot^t}$), and after it the trace is beyond the decision frontier ($\models_{\top^p}$ or $\models_{\bot^p}$). The fact that these five regions are pairwise disjoint and jointly exhaustive is established formally in Theorem 10.

Collapsed two-valued judgements. For downstream use (e.g., compliance checking), we collapse the five tight judgements into a two-valued view. For conservativeness, undecided prefixes are treated as violating, which prevents premature acceptance.

$$\pi \models_{\top} C \iff (\pi \models_{\top^t} C) \text{ or } (\pi \models_{\top^p} C),$$
$$\pi \models_{\bot} C \iff (\pi \models_{?} C) \text{ or } (\pi \models_{\bot^t} C) \text{ or } (\pi \models_{\bot^p} C).$$

Exactly one of π sat C or $\pi \models_{\bot} C$ holds for every trace π and contract C. This conservative collapse treats undecided prefixes as *violating* (no premature acceptance) while still preserving the tight moment of satisfaction.

Literal Tight Semantics

Literals ℓ are decided in a single synchronous step. We therefore interpret them on a single event word $\langle A \rangle$ with $A \in \Gamma$ (the set of actions that occurred in one period).

We present the literal clauses for party $p = 1$; the case $p = 2$ is symmetric (by swapping $(\cdot)^{(1)}$ and $(\cdot)^{(2)}$).

Intuitively, for party 1: (i) an *obligation* $\mathbf{O}_1(a)$ requires the joint execution of $a^{(1)}$ and $a^{(2)}$; (ii) a *prohibition* $\mathbf{F}_1(a)$ is satisfied precisely when that joint execution does not occur; and (iii) a *power* $\mathbf{P}_1(a)$ requires that whenever party 1 attempts $a^{(1)}$, party 2 simultaneously supports it with $a^{(2)}$.

Definition 47 (Literal tight satisfaction)**.** *For the empty trace, we have:*

$$\langle - \rangle \models_{?} \ell \quad \textit{for any literal } \ell.$$

Literals are decided on a single synchronous step. We define their semantics on a single event word $\langle A \rangle$ ($A \in \Gamma$) and the empty word $\langle - \rangle$.

$$\langle - \rangle \models_{?} \ell \quad \textit{for any literal } \ell.$$

Tight satisfaction. *For a single event word $\langle A \rangle$:*

$$\langle A \rangle \models_{\top^t} \top \quad \stackrel{\text{def}}{=} \quad \textit{true}.$$
$$\langle A \rangle \models_{\top^t} \bot \quad \stackrel{\text{def}}{=} \quad \textit{false}.$$
$$\langle A \rangle \models_{\top^t} \mathbf{O}_1(a) \quad \stackrel{\text{def}}{=} \quad \{a^{(1)}, a^{(2)}\} \subseteq A.$$
$$\langle A \rangle \models_{\top^t} \mathbf{F}_1(a) \quad \stackrel{\text{def}}{=} \quad \{a^{(1)}, a^{(2)}\} \not\subseteq A.$$
$$\langle A \rangle \models_{\top^t} \mathbf{P}_1(a) \quad \stackrel{\text{def}}{=} \quad \textit{if } (a^{(1)} \in A) \textit{ then } (a^{(2)} \in A).$$

Tight violation. *For a single event word $\langle A \rangle$:*

$$\begin{aligned}
\langle A \rangle &\models_{\perp^t} \top &&\stackrel{\text{def}}{=} \textit{false}.\\
\langle A \rangle &\models_{\perp^t} \bot &&\stackrel{\text{def}}{=} \textit{true}.\\
\langle A \rangle &\models_{\perp^t} \mathbf{O}_1(a) &&\stackrel{\text{def}}{=} \{a^{(1)}, a^{(2)}\} \not\subseteq A.\\
\langle A \rangle &\models_{\perp^t} \mathbf{F}_1(a) &&\stackrel{\text{def}}{=} \{a^{(1)}, a^{(2)}\} \subseteq A.\\
\langle A \rangle &\models_{\perp^t} \mathbf{P}_1(a) &&\stackrel{\text{def}}{=} (a^{(1)} \in A) \textit{ and } (a^{(2)} \notin A).
\end{aligned}$$

Example 54 (Literal satisfaction and violation)**.** *Let $A = \{a^{(1)}, a^{(2)}, b^{(2)}\}$ be the joint actions in one period. Then*

$$\langle A \rangle \models_{\top^t} \mathbf{O}_1(a), \qquad \langle A \rangle \models_{\top^t} \mathbf{P}_1(a), \qquad \langle A \rangle \models_{\top^t} \mathbf{F}_2(b).$$

Let $A' = \{a^{(1)}, b^{(1)}, b^{(2)}\}$. Then

$$\langle A' \rangle \models_{\perp^t} \mathbf{P}_1(a) \quad \textit{(since } a^{(2)} \notin A'\textit{)}, \qquad \langle A' \rangle \models_{\top^t} \mathbf{F}_2(a) \quad \textit{(no joint } a^{(1)} \textit{ and } a^{(2)} \textit{ occurs).}$$

Also, $\langle A' \rangle \models_{\top^t} \mathbf{P}_2(a)$ holds vacuously since $a^{(2)} \notin A'$.

Two-event trace. *Consider $\langle A, A' \rangle$. Since literals are decided at the first letter, the overall collapsed verdict follows from $\langle A \rangle$ are post satisfaction or post violation:*

$$\begin{aligned}
\langle A, A' \rangle &\models_{\top^p} \mathbf{O}_1(a),\\
\langle A, A' \rangle &\models_{\top^p} \mathbf{P}_1(a),\\
\langle A, A' \rangle &\models_{\perp^p} \mathbf{P}_2(b).
\end{aligned}$$

Binary Contract Operators Tight Semantics

Binary contract operators combine two contracts into structured compositions that capture parallel, sequential, or conditional behavior. In TACNL we use

$$\mathsf{op} \in \{\wedge, \ ; \ , \blacktriangleright\},$$

where $\wedge$ enforces *both* components, ; demands *first C then C'*, and $\blacktriangleright$ means "if C fails, *repair* with C'."

Reading guide (tight view). All clauses below are tight: they identify the *first decisive point* where satisfaction or violation becomes determined. For conjunction, the decisive point for satisfaction is the latter of the two individual successes; for violation, the first conjunct that fails. For sequencing, a split index k witnesses that C succeeds before C' is checked. Reparation, on the other hand, requires that either C succeeds directly, or, at the first tight violation of C, the repair C' must succeed on the remainder.

Definition 48 (Binary Contract Operators)**.** *Let π be a finite trace over $\Gamma = 2^{\Sigma}$ with $s = |\pi|$. We use the notation $\pi_k = \pi[1,k]$ for the prefix of length k, and $\pi^k = \pi[k+1,s]$ for the suffix after k.*

Conjunction $(C \wedge C')$

$$\pi \models_{\top^t} C \wedge C' \quad \stackrel{\text{def}}{=} \quad \exists k,k' \in [1,s] : \pi_k \models_{\top^t} C \text{ and } \pi_{k'} \models_{\top^t} C' \text{ and } s = \max(k,k'),$$

$$\pi \models_{\bot^t} C \wedge C' \quad \stackrel{\text{def}}{=} \quad (\pi \models_{\bot^t} C \text{ or } \pi \models_{\bot^t} C') \text{ and } \forall j \in [1,s-1] : \neg(\pi_j \models_{\bot^t} (C \wedge C')),$$

Sequence $(C \,;\, C')$

$$\pi \models_{\top^t} C;C' \quad \stackrel{\text{def}}{=} \quad \exists k \in [1,s-1] : \pi_k \models_{\top^t} C \text{ and } \pi^k \models_{\top^t} C',$$

$$\pi \models_{\bot^t} C;C' \quad \stackrel{\text{def}}{=} \quad \pi \models_{\bot^t} C \text{ or } \exists k \in [1,s-1] : \pi_k \models_{\top^t} C \text{ and } \pi^k \models_{\bot^t} C',$$

Reparation $(C \blacktriangleright C')$

$$\pi \models_{\top^t} C \blacktriangleright C' \quad \stackrel{\text{def}}{=} \quad \pi \models_{\top^t} C \text{ or } \exists k \in [1,s-1] : \pi_k \models_{\bot^t} C \text{ and } \pi^k \models_{\top^t} C',$$

$$\pi \models_{\bot^t} C \blacktriangleright C' \quad \stackrel{\text{def}}{=} \quad \exists k \in [1,s-1] : \pi_k \models_{\bot^t} C \text{ and } \pi^k \models_{\bot^t} C'.$$

Semantics summary. *Conjunction* succeeds once both parts succeed (possibly at different times); its decisive index is the latter of the two. It fails as soon as either part fails. *Sequence* requires a witness split k: first C succeeds on $[1,k]$, then C' on $[k+1,s]$. *Reparation* allows C' to take over at the first violation of C; overall success means either direct success of C or a violation-then-repair pattern.

Lemma 31 (Local disjointness of tight satisfaction and violation for binary operators)**.** *Let C, C' be contracts and let π be a finite trace. Assume the following induction hypotheses hold for the subcontracts:*

$$\neg\big(\pi \models_{\top^t} C \text{ and } \pi \models_{\bot^t} C\big) \qquad \text{and} \qquad \neg\big(\pi \models_{\top^t} C' \text{ and } \pi \models_{\bot^t} C'\big),$$

and, moreover, the same disjointness holds for every suffix π^k in place of π. Then, for each binary operator $\mathsf{op} \in \{\wedge, \,;\, , \blacktriangleright\}$ we have

$$\neg\big(\pi \models_{\top^t} (C \;\mathsf{op}\; C') \text{ and } \pi \models_{\bot^t} (C \;\mathsf{op}\; C')\big).$$

Proof. We prove the three cases by contradiction, using Definition 48. We also use Lemma 34 to rule out an opposite frontier on a strict prefix once a tight verdict holds on the full trace.

Case $\wedge$. Assume $\pi \models_{\top^t} (C \wedge C')$ and $\pi \models_{\bot^t} (C \wedge C')$. From $\pi \models_{\top^t} (C \wedge C')$ there exist $k, k' \in [1, s]$ such that $\pi_k \models_{\top^t} C$, $\pi_{k'} \models_{\top^t} C'$, and $s = \max(k, k')$. From $\pi \models_{\bot^t} (C \wedge C')$ we obtain $\pi \models_{\bot^t} C$ or $\pi \models_{\bot^t} C'$.

If $\pi \models_{\bot^t} C$, then: (i) if $k = s$, we have $\pi \models_{\top^t} C$ and $\pi \models_{\bot^t} C$, contradicting the disjointness hypothesis for C; (ii) if $k < s$, then π_k is a strict prefix of π with $\pi_k \models_{\top^t} C$, contradicting Lemma 34(2) instantiated with C (since $\pi \models_{\bot^t} C$ forbids any $j < s$ with $\pi_j \models_{\top^t} C$). The case $\pi \models_{\bot^t} C'$ is symmetric.

Case ;. Assume $\pi \models_{\top^t} (C;C')$ and $\pi \models_{\bot^t} (C;C')$. From satisfaction there exists $k \in [1, s-1]$ such that $\pi_k \models_{\top^t} C$ and $\pi^k \models_{\top^t} C'$. From violation either (i) $\pi \models_{\bot^t} C$ or (ii) there exists $m \in [1, s-1]$ such that $\pi_m \models_{\top^t} C$ and $\pi^m \models_{\bot^t} C'$.

In case (i), $\pi \models_{\bot^t} C$ contradicts Lemma 34(2) for C because π_k is a strict prefix with $\pi_k \models_{\top^t} C$.

In case (ii), tight satisfaction of C is a frontier event along prefixes of a fixed trace, so $\pi_k \models_{\top^t} C$ and $\pi_m \models_{\top^t} C$ imply $k = m$. Hence, the same suffix π^k both tightly satisfies and tightly violates C', namely $\pi^k \models_{\top^t} C'$ and $\pi^k \models_{\bot^t} C'$, contradicting the suffix-disjointness hypothesis for C'.

Case $\blacktriangleright$. Assume $\pi \models_{\top^t} (C \blacktriangleright C')$ and $\pi \models_{\bot^t} (C \blacktriangleright C')$. From $\pi \models_{\bot^t} (C \blacktriangleright C')$ there exists $k \in [1, s-1]$ such that $\pi_k \models_{\bot^t} C$ and $\pi^k \models_{\bot^t} C'$. From $\pi \models_{\top^t} (C \blacktriangleright C')$ either (a) $\pi \models_{\top^t} C$ or (b) there exists $m \in [1, s-1]$ such that $\pi_m \models_{\bot^t} C$ and $\pi^m \models_{\top^t} C'$.

In case (a), $\pi \models_{\top^t} C$ and the strict-prefix violation $\pi_k \models_{\bot^t} C$ contradict Lemma 34(1) instantiated with C.

In case (b), tight violation of C is also a frontier event along prefixes of a fixed trace, so $\pi_k \models_{\bot^t} C$ and $\pi_m \models_{\bot^t} C$ imply $k = m$. Hence, the same suffix π^k both violates and satisfies C', namely $\pi^k \models_{\bot^t} C'$ and $\pi^k \models_{\top^t} C'$, contradicting the suffix-disjointness hypothesis for C'.

Thus in all three cases we reach a contradiction, so $\neg\big(\pi \models_{\top^t} (C \text{ op } C') \text{ and } \pi \models_{\bot^t} (C \text{ op } C')\big)$. □

Example 55 (Tight satisfaction and violation for $(C_2 \wedge C_3)$)**.** *We reuse the collaboration alphabet*

$$\Sigma_C = \{\mathsf{PAY\_R},\ \mathsf{PAY\_F},\ \mathsf{OCC},\ \mathsf{Notif\_R},\ \mathsf{Maint\_P}\},$$

and recall

$$C_2 := \mathbf{P}_1(\mathsf{OCC}), \qquad C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \ \blacktriangleright\ \mathbf{O}_1(\mathsf{PAY\_F}).$$

Tight satisfaction (the longest prefix). *Consider the trace*

$$\pi_{sat} = \langle A_1, A_2 \rangle, \qquad A_1 = \{\mathsf{OCC}^{(2)}\}, \quad A_2 = \{\mathsf{PAY\_F}^{(1)}, \mathsf{PAY\_F}^{(2)}\}.$$

Then $\langle A_1 \rangle \models_{\top^t} C_2$ *(vacuously, since* $\mathsf{OCC}^{(1)} \notin A_1$ *and no unsupported attempt occurs), and* $\pi_{sat} \models_{\top^t} C_3$ *(the rent was not paid in month 1, but the reparation clause succeeds at* $t{=}1$*). Hence, by conjunction,* $\pi_{sat} \models_{\top^t} (C_2 \wedge C_3)$ *at the longest decisive prefix.*

Tight satisfaction (the shortest prefix). *For the single-event trace*

$$\langle A'_1 \rangle \quad \textit{with} \quad A'_1 = \{\mathsf{OCC}^{(2)}, \mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\},$$

we have $\langle A'_1 \rangle \models_{\top^t} C_2$ *and* $\langle A'_1 \rangle \models_{\top^t} \mathbf{O}_1(\mathsf{PAY\_R})$*, so both conjuncts hold in month 1:*

$$\langle A'_1 \rangle \models_{\top^t} (C_2 \wedge C_3), \quad \textit{and any extension } \langle A'_1, A_2 \rangle \textit{ yields } \models_{\top^p} (C_2 \wedge C_3).$$

Tight violation. *Now consider*

$$\pi_{viol} = \langle A_1, A_2 \rangle, \qquad A_1 = \{\mathsf{OCC}^{(1)}\}, \quad A_2 = \emptyset.$$

In the first month, $\langle A_1 \rangle \models_{\top^t} C_2$ *and* $\langle A_1 \rangle \models_{\bot^t} \mathbf{O}_1(\mathsf{PAY\_R})$*, while* $\langle A_1 \rangle \models_{?} C_3$ *since the reparation in* $C_3 = \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$ *has not yet been tested. At* $t{=}1$*,* $\pi_{viol} \models_{\bot^t} C_3$ *(as* $\pi[1,1] \models_{\bot^t} \mathbf{O}_1(\mathsf{PAY\_R})$ *and* $\pi[2,2] \models_{\bot^t} \mathbf{O}_1(\mathsf{PAY\_F})$*). Thus, the overall violation arises from* C_3*, and by conjunction,* $\pi_{viol} \models_{\bot^t} (C_2 \wedge C_3)$*.*

This shows that $(C_2 \wedge C_3)$ *satisfies either immediately when both conjuncts hold, or later when a reparation compensates for a missed rent, while violation arises when both payment and its repair fail.*

Repetition Contracts Tight Semantics

Definition 49 (Repetition Contracts)**.** *Let* π *be a finite trace over the event alphabet* $\Gamma = 2^{\Sigma}$*, with* $s = |\pi|$*. For* $k \in \{1, \ldots, s-1\}$*, we denote by* $\pi_k := \pi[1,k]$ *the prefix of length* k*, and by* $\pi^k := \pi[k+1, s]$ *the corresponding suffix. Let* $n \in \mathbb{N}^*$ *be a strictly positive natural number.*

The semantics for repetition contracts are inductively defined as follows:

$$\pi \models_{\top^t} C^n \quad \stackrel{\text{def}}{=} \quad \big(\text{if } n > 1 \text{ then } \pi \models_{\top^t} C;C^{n-1}\big) \text{ and } \big(n = 1 \Rightarrow \pi \models_{\top^t} C\big),$$

$$\pi \models_{\bot^t} C^n \quad \stackrel{\text{def}}{=} \quad \big(\pi \models_{\bot^t} C\big) \text{ or } \big(\exists k \in [1, s-1], \exists 1 \leq m < n : \pi_k \models_{\top^t} C^m \text{ and } \pi^k \models_{\bot^t} C\big),$$

$$\pi \models_{\top^t} \mathbf{Rep}(C) \quad \stackrel{\text{def}}{=} \quad \text{false},$$

$$\pi \models_{\bot^t} \mathbf{Rep}(C) \quad \stackrel{\text{def}}{=} \quad \exists n \in \mathbb{N}^* : \pi \models_{\bot^t} C^n.$$

Intuition. Repetition contracts express the iterative enforcement of a subcontract. The finite form C^n requires C to hold n times in sequence, each instance starting immediately after the previous one completes. The satisfaction condition unfolds recursively: a trace satisfies C^n if it can be decomposed into a prefix where C holds, followed by a suffix that satisfies C^{n-1}. A violation occurs either when the first occurrence of C fails, or when some later repetition cannot be fulfilled after a previously satisfied segment (captured by the split $\pi_k \models_{\top^t} C^m$ and $\pi^k \models_{\bot^t} C$). Hence, C^n behaves as a *sequential chain* of responsibilities and rights, and any broken link invalidates the entire chain.

The infinite form $\mathbf{Rep}(C)$ captures *unbounded repetition*. Since finite traces cannot exhibit infinite iteration, $\mathbf{Rep}(C)$ is never fully satisfied (false under tight semantics); it is only meaningful with respect to violation: a trace violates $\mathbf{Rep}(C)$ once it violates one of its finite unfolding C^n. Intuitively, $\mathbf{Rep}(C)$ models *renewable or continuing* contracts such as subscriptions or recurring payments, where each cycle restarts the same normative condition indefinitely.

Lemma 32 (Local disjointness for repetition contracts)**.** *Let C be a contract, $n \in \mathbb{N}^*$, and let π be a finite trace. Assume the induction hypothesis that for every finite trace $\Pi_{\min}$ we have $\neg(\Pi_{\min} \models_{\top^t} C \text{ and } \Pi_{\min} \models_{\bot^t} C)$, and moreover the same disjointness holds for every suffix $\Pi^k_{\min}$ in place of $\Pi_{\min}$. Then:*

$$\neg\big(\pi \models_{\top^t} C^n \text{ and } \pi \models_{\bot^t} C^n\big) \qquad \text{and} \qquad \neg\big(\pi \models_{\top^t} \mathbf{Rep}(C) \text{ and } \pi \models_{\bot^t} \mathbf{Rep}(C)\big).$$

Proof. We prove the two claims.

Finite repetition C^n. We argue by induction on n. For $n = 1$, we have $C^1 \equiv C$ by definition, hence the claim follows from the hypothesis. For $n > 1$, the definition gives $\pi \models_{\top^t} C^n$ iff $\pi \models_{\top^t} C;C^{n-1}$. Likewise, the violation clause for C^n can only arise either from an initial violation of C, or after some satisfied prefix where a later copy of C is violated. In either case, if we assume toward contradiction that $\pi \models_{\top^t} C^n$ and $\pi \models_{\bot^t} C^n$,

then we obtain a contradiction with (i) Lemma 31 for the sequence operator (applied to C and C^{n-1}), (ii) the induction hypothesis for C, and (iii) the induction hypothesis for C^{n-1} together with the suffix-disjointness assumption. Thus, $\neg(\pi \models_{\top^t} C^n \text{ and } \pi \models_{\bot^t} C^n)$ holds.

Unbounded repetition Rep(C). By definition, $\pi \models_{\top^t} \mathbf{Rep}(C)$ is *false* for every finite trace π. Hence, π cannot both tightly satisfy and tightly violate $\mathbf{Rep}(C)$. □

Regular Expression Binary Operator Semantics

Contracts guarded by regular expressions specify that a normative condition becomes active only after the trace matches a given regular pattern. Such patterns, written *re*, are interpreted over the letter alphabet $\Gamma = 2^{\Sigma}$ introduced above. They act as *temporal triggers* that delimit where an obligation, prohibition, or reparation clause starts to apply.

Two guarded forms are distinguished:

- The *triggered contract* $\langle\langle re \rangle\rangle C$, which activates C as soon as a prefix of the trace matches *re*.
- The *guarded contract* $\lceil re \rceil C$, which restricts C to hold only while the trace remains within the language induced by *re*.

The first captures temporal activation ("after the trigger, C must hold"), the second conditional persistence ("as long as *re* remains possible, C must hold").

Definition 50 (Triggered and Guarded Contracts)**.** *Let π be a finite trace over $\Gamma = 2^{\Sigma}$ with $s = |\pi|$.*

$$\begin{aligned}
\pi \models_{\top^t} \langle\langle re \rangle\rangle C &\stackrel{\text{def}}{=} \pi \models_{\bot^t} re \text{ or } \big(\exists k \in [1, s-1] : \pi_k \models_{\top^t} re \text{ and } \pi^k \models_{\top^t} C\big),\\
\pi \models_{\bot^t} \langle\langle re \rangle\rangle C &\stackrel{\text{def}}{=} \exists k \in [1, s-1] : \pi_k \models_{\top^t} re \text{ and } \pi^k \models_{\bot^t} C,\\
\pi \models_{\top^t} \lceil re \rceil C &\stackrel{\text{def}}{=} \big(\pi \models_{\bot^t} re \text{ and } \pi \models_{Cl} C\big) \text{ or } \big(\pi \models_{Cl} re \text{ and } \pi \models_{\top^t} C\big),\\
\pi \models_{\bot^t} \lceil re \rceil C &\stackrel{\text{def}}{=} \pi \models_{Cl} re \text{ and } \pi \models_{\bot^t} C.
\end{aligned}$$

Where $\pi \models_{Cl} X$ abbreviates $(\pi \models_{?} X \text{ or } \pi \models_{\top^t} X)$.

Lemma 33 (Local disjointness for triggered and guarded contracts)**.** *Let re be a regular expression over Γ and let C be a contract. Assume disjointness for both components, namely for every finite trace $\Pi_{\min}$:*

$$\neg(\Pi_{\min} \models_{\top^t} re \text{ and } \Pi_{\min} \models_{\bot^t} re) \quad \text{and} \quad \neg(\Pi_{\min} \models_{\top^t} C \text{ and } \Pi_{\min} \models_{\bot^t} C),$$

and assume the same disjointness holds for every suffix $\Pi^k_{\min}$ *in place of* $\Pi_{\min}$. *Then for every finite trace* π*:*

$$\neg\big(\pi \models_{\top^t} \langle\langle re\rangle\rangle C \text{ and } \pi \models_{\bot^t} \langle\langle re\rangle\rangle C\big) \quad \text{and} \quad \neg\big(\pi \models_{\top^t} \lceil re\rceil C \text{ and } \pi \models_{\bot^t} \lceil re\rceil C\big).$$

Proof. We treat the two constructors.

Triggered $\langle\langle re\rangle\rangle C$**.** Assume toward contradiction that $\pi \models_{\top^t} \langle\langle re\rangle\rangle C$ and $\pi \models_{\bot^t} \langle\langle re\rangle\rangle C$. From the violation clause, there exists $k \in [1, s-1]$ such that $\pi_k \models_{\top^t} re$ and $\pi^k \models_{\bot^t} C$.

From the satisfaction clause, either (a) $\pi \models_{\bot^t} re$ or (b) there exists $m \in [1, s-1]$ such that $\pi_m \models_{\top^t} re$ and $\pi^m \models_{\top^t} C$.

Case (a): We have $\pi_k \models_{\top^t} re$ (from violation) and $\pi \models_{\bot^t} re$ (from assumption). By Lemma 34 (Mutual Prefix Exclusion applied to *re*), if a prefix π_k tightly satisfies *re*, the full trace π (which is an extension of π_k) cannot tightly violate *re*. Contradiction.

Case (b): We have $\pi_k \models_{\top^t} re$ and $\pi_m \models_{\top^t} re$. By Lemma 34 applied to *re*, there is at most one prefix of π that tightly satisfies *re*. Thus, $k = m$. This implies we have both $\pi^k \models_{\bot^t} C$ and $\pi^k \models_{\top^t} C$ on the *same* suffix. This contradicts the disjointness hypothesis for C on the suffix π^k.

Hence, π cannot both tightly satisfy and tightly violate $\langle\langle re\rangle\rangle C$.

Guarded $\lceil re\rceil C$**.** Assume toward contradiction that $\pi \models_{\top^t} \lceil re\rceil C$ and $\pi \models_{\bot^t} \lceil re\rceil C$. By Definition 50, violation means $\pi \models_{Cl} re$ and $\pi \models_{\bot^t} C$. Satisfaction means either (i) $\pi \models_{\bot^t} re$ and $\pi \models_{Cl} C$, or (ii) $\pi \models_{Cl} re$ and $\pi \models_{\top^t} C$.

Case (ii): This yields $\pi \models_{Cl} re$ (consistent) but $\pi \models_{\top^t} C$ and $\pi \models_{\bot^t} C$. This contradicts the disjointness hypothesis for C.

Case (i): This yields $\pi \models_{Cl} re$ and $\pi \models_{\bot^t} re$. Expanding $\models_{Cl}$, this means ($\pi \models_? re$ or $\pi \models_{\top^t} re$) and $\pi \models_{\bot^t} re$. By the disjointness hypothesis for *re*, $\models_{\top^t}$ excludes $\models_{\bot^t}$. By Lemma 34, $\models_?$ excludes $\models_{\bot^t}$. Thus, Case (i) is impossible.

Thus guarded contracts are also disjoint. □

Example 56 (Triggered and guarded contracts)**.** *Let the collaboration alphabet be*

$$\Sigma_C = \{\mathsf{PAY\_R},\ \mathsf{PAY\_F},\ \mathsf{OCC},\ \mathsf{Notif\_R},\ \mathsf{Notif\_T},\ \mathsf{Maint\_P}\}.$$

(a) Triggered contract. *Clause C_4 specifies that when the tenant requests a repair, the landlord must perform it within the following period:*

$$C_4 := \langle\langle\{\mathsf{Notif\_R}^{(1)}\}\rangle\rangle \mathbf{O}_2(\mathsf{Maint\_P}).$$

Tight satisfaction.

$$\pi_{\mathsf{sat}} = \langle A_1, A_2\rangle, \text{ with } A_1 = \{\mathsf{Notif\_R}^{(1)}\} \text{ and } A_2 = \{\mathsf{Maint\_P}^{(1)}, \mathsf{Maint\_P}^{(2)}\}.$$

In the first month, the trigger $\mathsf{Notif\_R}^{(1)}$ *occurs, activating the repair obligation. In month 2, the landlord performs* $\mathsf{Maint\_P}^{(1,2)}$*, thus* $\pi_{\mathsf{sat}} \models_{\top^t} C_4$.

Tight violation.

$$\pi_{\mathsf{viol}} = \langle A_1, A_2'\rangle, \text{ with } A_1 = \{\mathsf{Notif\_R}^{(1)}\} \text{ and } A_2' = \emptyset.$$

The trigger in month 1, but the obligation is unfulfilled: $\pi_{\mathsf{viol}} \models_{\bot^t} C_4$.

(b) Guarded repetition. *To limit repetition to the occupancy period, combine guard and repetition:*

$$C_9 := \lceil \Gamma^+ ; \{\mathsf{Notif\_T}^{(1)}\} ; \Gamma^3 \rceil \mathbf{Rep}(\mathbf{O}_1(\mathsf{PAY\_R})).$$

The guard pattern $\Gamma^+;\{\mathsf{Notif\_T}^{(1)}\};\Gamma^3$ *means "for any non-empty prefix up to the termination notice* $\mathsf{Notif\_T}^{(1)}$*, and for at most three additional steps afterward." Within this region, the obligation to pay rent repeats. Once* $\mathsf{Notif\_T}^{(1)}$ *occurs, the duty remains for three more periods, and the contract is satisfied at* t=1+3=4.

Tight satisfaction.

$$\pi_{\mathsf{sat}} = \langle A_1, A_2, A_3, A_4, A_5\rangle, \quad \begin{aligned} A_1 &= \{\mathsf{OCC}^{(1)}, \mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\}, \\ A_2 &= \{\mathsf{Notif\_T}^{(1)}, \mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\}, \\ A_3 &= A_4 = A_5 = \{\mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\}. \end{aligned}$$

The guard is satisfied in month 5 and the payments were all successful, hence $\pi_{\mathsf{sat}} \models_{\top^t} C_9$.

Tight violation.

$$\pi_{\mathsf{viol}} = \langle A_1, A_2, A_3, A_4, A_5\rangle, \quad \begin{aligned} A_1 &= \{\mathsf{OCC}^{(1)}, \mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\}, \\ A_2 &= \{\mathsf{Notif\_T}^{(1)}, \mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\}, \\ A_3 &= \{\mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\}, \\ A_4 &= \emptyset, \\ A_5 &= \{\mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\}. \end{aligned}$$

A missing payment in the fourth month breaks the repetition duty while the guard still holds, so $\pi_{\mathsf{viol}} \models_{\bot^t} C_9$.

Coherence of the Forward-Looking Contract Satisfaction Semantics

Coherence requires that for any fixed contract and trace, there is never more than one decisive verdict. A trace cannot both tightly satisfy and tightly violate the same contract on different prefixes, since this would yield two incompatible outcomes for a single execution. Forward semantics must therefore rule out situations where tight satisfaction appears on one prefix and tight violation appears on another prefix of the same trace. Ensuring this exclusion makes the decisive point unique, which is required to justify every verdict from $\mathbb{V}_5$. The next lemma states this exclusion precisely by showing that the two frontiers cannot arise on distinct prefixes of the same trace.

Lemma 34 (Mutual prefix exclusion tight satisfaction and violation)**.** *For every contract C in* TACNL *and every finite trace π, the tight satisfaction and tight violation forward semantics are mutually exclusive, that is:*

1. No earlier tight violation at or after tight satisfaction.
   If $\pi \models_{\top^t} C$ *then* $\nexists j < |\pi| : \pi[1,j] \models_{\bot^t} C$
2. No earlier tight satisfaction at or after tight violation.
   if $\pi \models_{\bot^t} C$ *then* $\nexists j < |\pi| : \pi[1,j] \models_{\top^t} C$

Proof sketch. By structural induction on the syntactical structure of C.

Base case: literals. By Definition 47, a literal is decided on a single letter: $\langle A \rangle \models_{\top^t} \ell$ iff the letter constraint holds, and $\langle A \rangle \models_{\bot^t} \ell$ iff it does not. These are complements on that step, so the two implications are immediate, and uniqueness follows.

Inductive hypotheses. Assume the theorem holds for subcontracts as needed below. We use:

$$\text{(IH-}C\text{-sat)} \quad \forall \pi \left(\pi \models_{\top^t} C \Rightarrow \forall j < |\pi| : \neg(\pi[1,j] \models_{\bot^t} C) \right),$$
$$\text{(IH-}C\text{-viol)} \quad \forall \pi \left(\pi \models_{\bot^t} C \Rightarrow \forall j < |\pi| : \neg(\pi[1,j] \models_{\top^t} C) \right),$$

and similarly (IH-C'-sat) and (IH-C'-viol) when a second operand C' is present; for regex guards re we use the same two clauses with re in place of C.

Conjunction $C \wedge C'$. By Definition 48, tight satisfaction requires first successes at some k, k' with decisive index $j^\star = \max\{k, k'\}$. For every $j < j^\star$, either $j < k$ or $j < k'$ holds, hence by (IH-C-sat) and (IH-C'-sat) neither $\pi[1,j] \models_{\bot^t} C$ nor $\pi[1,j] \models_{\bot^t} C'$ holds. Since a tight violation of a conjunction is a tight violation of the conjunction, no $j < j^\star$ violates $C \wedge C'$. This proves the first implication. For the second, if some prefix tightly violates a conjunct, then by (IH-C-viol) or (IH-C'-viol) no earlier prefix tightly satisfies that conjunct, hence, no earlier prefix tightly satisfies the conjunction.

Sequence $C;C'$. By Definition 48, tight satisfaction needs a split k with $\pi[1,k] \models_{\top^t} C$ and $\pi[k+1,|\pi|] \models_{\top^t} C'$. For any $j \le k$, (IH-C-sat) forbids $\pi[1,j] \models_{\bot^t} C$; for any $j > k$, (IH-C'-sat) applied to the suffix forbids $\models_{\bot^t} C'$ before its own decisive point. A tight violation of $C;C'$ before satisfaction is either a violation of C before k or a violation of C' after k, both excluded. The dual implication follows from (IH-C-viol) and (IH-C'-viol).

Reparation $C \blacktriangleright C'$. By Definition 48, either C succeeds, or else at the first tight violation index k of C the repair C' must succeed on π^k. In the first branch (IH-C-sat), it excludes earlier violations. In the second branch, (IH-C-viol) gives minimal property of the failure point of C, and (IH-C'-sat) on the suffix excludes earlier failure of the composite before its tight success. The dual implication is symmetric, using (IH-C-viol) and (IH-C'-viol).

Finite repetition C^n. Unfold $C^n \equiv C;(C^{n-1})$ and argue by a secondary induction on n, using the sequence case and the induction hypotheses for C and C^{n-1}.

Unbounded repetition $\mathbf{Rep}(C)$. Under tight semantics $\mathbf{Rep}(C)$ never tightly satisfies and tightly violates iff some finite unrolling C^m tightly violates. The two implications reduce to the finite case above.

Triggered $\langle re \rangle C$. By Definition 50, either re is violated and the contract tightly satisfies vacuously, or there is a first k with $\pi_k \models_{\top^t} re$ and then the suffix must satisfy C. In the vacuous branch, (IH-re-viol) forbids any earlier tight satisfaction of re, so there is no earlier tight violation of the composite. In the active branch, the first match index k is minimal by (IH-re-sat); before k the composite is undecided, and after k we apply (IH-C-sat)/(IH-C-viol) on the suffix to obtain both implications.

Guarded $[re]C$. While π *closes* re (that is, π is in the open region for re), any tight or post failure of C yields a tight or post failure of the composite. Once re becomes impossible, the composite satisfies provided C has not failed. Combine (IH-re-sat) and (IH-re-viol) with (IH-C-sat) and (IH-C-viol), and the guarded case table in Definition 50, to derive the two implications.

All constructors preserve the two "no-backtrack" properties; hence, the claim holds for all C. □

Lemma 35 (Constructor-wise disjointness of the tight frontiers)**.** *For every contract C in* TACNL *and every finite trace π, we have*

$$\neg\big(\pi \models_{\top^t} C \text{ and } \pi \models_{\bot^t} C\big).$$

Proof. By structural induction on the syntax of C.

Base: literals. Disjointedness holds by Definition 47, since on a single letter the satisfaction and violation clauses are complementary.

Inductive steps. For binary constructors $C_1 \wedge C_2$, $C_1;C_2$, and $C_1 \blacktriangleright C_2$, the claim follows from Lemma 31 and the induction hypotheses for C_1 and C_2 (including the required suffix form). For repetition constructors D^n and $\mathbf{Rep}(D)$, the claim follows from Lemma 32 and the induction hypothesis for D. For triggered and guarded constructors $\langle\langle re\rangle\rangle D$ and $\lceil re\rceil D$, the claim follows from Lemma 33 together with the induction hypotheses for re and D. No other constructors exist. □

Theorem 10 (Consistency of the forward-looking five tight semantics for TACNL)**.** *The five forward satisfaction relations $\{\models_?, \models_{\top^t}, \models_{\bot^t}, \models_{\top^p}, \models_{\bot^p}\}$ for* TACNL *are pairwise disjoint and jointly exhaustive.*

Proof. Fix a finite trace π and contract C. We reason point-wise on prefixes of π (as defined in Section Section 2.2) and lift the result to the trace-level relations via Definition 46.

By Lemma 34, along a fixed trace the two frontiers cannot both occur on (possibly different) prefixes: if some prefix tightly satisfies C, then no prefix tightly violates C, and conversely. Moreover, constructor-wise disjointness holds point wise for every prefix, that is, $\neg(\Pi_{\min} \models_{\top^t} C \text{ and } \Pi_{\min} \models_{\bot^t} C)$ for every prefix $\Pi_{\min}$. This is established formally in Lemma 35.

Now consider any prefix position j with $1 \le j < |\pi|$ and view the corresponding trace prefix $\pi[1,j]$. By Definition 46, for a given prefix $\Pi_{\min} := \pi[1,j]$ exactly one of the following holds: (i) $\Pi_{\min} \models_? C$ (no earlier prefix of $\Pi_{\min}$ is decisive), (ii) $\Pi_{\min} \models_{\top^t} C$, (iii) $\Pi_{\min} \models_{\bot^t} C$, (iv) $\Pi_{\min} \models_{\top^p} C$ (some earlier prefix of $\Pi_{\min}$ tightly satisfies C), or (v) $\Pi_{\min} \models_{\bot^p} C$ (some earlier prefix of $\Pi_{\min}$ tightly violates C). These cases are mutually exclusive by the local disjointness results above together with Lemma 34, which prevents mixing satisfaction-frontiers and violation-frontiers along the same trace.

Therefore, the five relations $\{\models_?, \models_{\top^t}, \models_{\bot^t}, \models_{\top^p}, \models_{\bot^p}\}$ are pairwise disjoint and jointly exhaustive on prefixes of π. Since π was arbitrary, the claim holds for all finite traces. □

Next step. After establishing the forward tight satisfaction semantics in TACNL and studied their soundness. We now move from the denotational clauses to an operational view: we construct the corresponding Moore-style monitors for the tight five-valued semantics and establish that their outputs coincide with the semantic judgements on every finite trace prefix.

4.6.2 Monitor Construction for Tight Contract Satisfaction

From semantics to monitors. The definitions in Section Subsection 4.6.1 specify, for each contract C, which trace prefixes are tight satisfaction and tight violation frontiers, and how the remaining verdicts are derived from them. In this section we turn these clauses into finite-state monitors that compute the same verdicts incrementally along a trace. The construction is Moore-style: each state summarizes exactly the information needed to determine the current five-valued verdict, and each new letter updates this summary by a deterministic transition. We prove that the monitor output agrees with the denotational semantics at every prefix, which enables algorithmic compliance checking and serves as the basis for the responsibility-aware extensions developed next.

Definition 51 (Tight satisfaction monitor)**.** *The* tight satisfaction monitor*, written* $\mathcal{M}^{TS}$*, is a Moore machine whose output alphabet is the five-valued verdict set* $\mathbb{V}_5$*. Formally,*

$$\mathcal{M}^{TS} = (Q, q_0, \Gamma, \mathbb{V}_5, \delta, \lambda_5),$$

where:

1. *The output alphabet formed by 5 letters is*

$$\mathbb{V}_5 = \{?, \top^t, \bot^t, \top^p, \bot^p\}.$$

2. *Q is the set of states and $q_0 \in Q$ is the initial state,*
3. *$\Gamma = 2^{\Sigma}$ is the input event alphabet,*
4. *$\delta : Q \times \Gamma \to Q$ is the transition function,*
5. *$\lambda_5 : Q \to \mathbb{V}_5$ is the state output function.*

The next definition introduces the construction for any contract into its tight satisfaction monitor. The construction is a function that maps each contract C to a tight satisfaction monitor that enforces it. The construction proceeds by structural induction on the syntax of C, and each operator in TACNL is matched by a corresponding monitor combination operator. Regular expression guards and triggers rely on the tight satisfaction monitor TSMC(re) introduced earlier in Definition 45.

Definition 52 (Tight Satisfaction Monitor Construction)**.** *The* tight satisfaction monitor construction *is a function defined on* TACNL*, written* $\mathsf{TSMC}(C)$*, that returns the tight satisfaction monitor for a contract* C *in* TACNL*. It is defined inductively on the structure of* C*:*

$$\mathsf{TSMC}(C) := \begin{cases} \mathsf{TSMC}_{lit}(\ell) & \text{if } C = \ell, \\ \mathsf{TSMC}_{\wedge}\big(\mathsf{TSMC}(C_1), \mathsf{TSMC}(C_2)\big) & \text{if } C = C_1 \wedge C_2, \\ \mathsf{TSMC}_{;}\big(\mathsf{TSMC}(C_1), \mathsf{TSMC}(C_2)\big) & \text{if } C = C_1 ; C_2, \\ \mathsf{TSMC}_{\blacktriangleright}\big(\mathsf{TSMC}(C_1), \mathsf{TSMC}(C_2)\big) & \text{if } C = C_1 \blacktriangleright C_2, \\ \mathsf{TSMC}_{trig}(\mathsf{re}, C') & \text{if } C = \langle\langle \mathsf{re} \rangle\rangle C', \\ \mathsf{TSMC}_{guard}(\mathsf{re}, C') & \text{if } C = \lceil \mathsf{re} \rceil C', \\ \mathsf{TSMC}_{nrep}\big(n, \mathsf{TSMC}(C')\big) & \text{if } C = (C')^n, \\ \mathsf{TSMC}_{Rep}\big(\mathsf{TSMC}(C')\big) & \text{if } C = \mathbf{Rep}(C') \,. \end{cases}$$

Where the tight monitor construction for regular expressions $\mathsf{TSMC}(\mathsf{re})$ *is already defined in Definition 45.*

Monitor correctness invariant. The monitor construction preserves the following invariant. For every contract C in TACNL, every finite trace π, and every prefix π_k, the state reached by the monitor $\mathsf{TSMC}(C)$ after reading π_k emits exactly the verdict prescribed by the five-valued tight semantics, that is:

$$\lambda_5\big(\delta(q_0, \pi_k)\big) \;=\; [\![\pi_k \vDash C]\!]_5 \,.$$

All operator-specific constructions are designed to preserve this invariant under synchronous product, redirection, and state elimination. Correctness lemmas below establish that the invariant holds inductively for each contract constructor.

We construct in the next subsections the tight satisfaction monitors compositionally from the syntax of contracts. The construction proceeds by structural induction on the contract and associates to each contract operator a corresponding monitor-combination operator. Conceptually, this follows the classical construction paradigm of Thompson for regular expressions, where complex behaviors are obtained by composing simpler automata. However, rather than constructing an intermediate nondeterministic automaton and determinizing it afterward, we perform the construction directly at the level of deterministic Moore machines. In particular, verdict propagation is built into the states and outputs of the monitor from the outset, so that tight satisfaction and violation are observed incrementally on prefixes of the trace. This avoids a separate determinization step and ensures that the monitor semantics coincides by construction with the five-valued tight semantics.

Construction for Literal Contracts

From Tight Semantics to 5-Valued Monitoring. The literal clauses above define one-step satisfaction and violation judgements for a single event word $\langle A \rangle$. We lift these clauses into a five-valued Moore machine whose outputs track the evolution of the tight verdicts over prefixes.

Definition 53 (Tight Satisfaction Monitor Construction for Literals)**.** *For a literal ℓ from* TACNL*, the* Tight Satisfaction monitor construction *for ℓ, written* $\mathsf{TSMC}_{lit}(\ell)$ *is defined as:*

$$\mathsf{TSMC}_{lit}(\ell) = (Q, q_0, \Gamma, \mathbb{V}_5, \delta, \lambda_5).$$

- $Q = \{q_0, q_s, q_v, q_{ps}, q_{pv}\}$*, with outputs* $\lambda_5(q_0) = ?$, $\lambda_5(q_s) = \top^t$, $\lambda_5(q_v) = \bot^t$, $\lambda_5(q_{ps}) = \top^p$, $\lambda_5(q_{pv}) = \bot^p$.
- $\Gamma = 2^{\Sigma}$ *is the event alphabet.*
- $\delta : Q \times \Gamma \to Q$ *is defined as follows:*

$$\begin{aligned}
&\textit{(1) tight transition:} && \delta(q_0, A) = \begin{cases} q_s & \textit{if } \langle A \rangle \models_{\top^t} \ell, \\ q_v & \textit{if } \langle A \rangle \models_{\bot^t} \ell; \end{cases} \\
&\textit{(2) post transitions:} && \delta(q_s, A) = q_{ps},\ \delta(q_v, A) = q_{pv}, \\
& && \delta(q_{ps}, A) = q_{ps},\ \delta(q_{pv}, A) = q_{pv}.
\end{aligned}$$

Hence, for every atomic literal, the machine emits ? at the initial state, switches to $\top^t$ or $\bot^t$ at the next state by consuming the event forming $\langle A \rangle$, and then permanently outputs $\top^p$ or $\bot^p$ for all remaining events. This Moore representation is equivalent to the tight semantics of Definition 47 but refines it with explicit prefix continuity.

Construction for Binary Contract Operators

For binary contract operators of the form C op C' with $\mathsf{op} \in \{\wedge,\ ;\ ,\ \blacktriangleright\}$, the monitor for the composite contract is obtained by combining the already constructed monitors $\mathsf{TSMC}(C)$ and $\mathsf{TSMC}(C')$. Each operator has its own monitor construction, defined below for conjunction, sequence, and reparation. We begin with the conjunction case.

Definition 54 (Tight Satisfaction Monitor Construction for Conjunction)**.** *Let C and C' be contracts in* TACNL*, and let*

$$\mathsf{TSMC}(C) = (Q, q_0, \Gamma, \mathbb{V}_5, \delta, \lambda_5) \quad \textit{and} \quad \mathsf{TSMC}(C') = (Q', q_0', \Gamma, \mathbb{V}_5, \delta', \lambda_5').$$

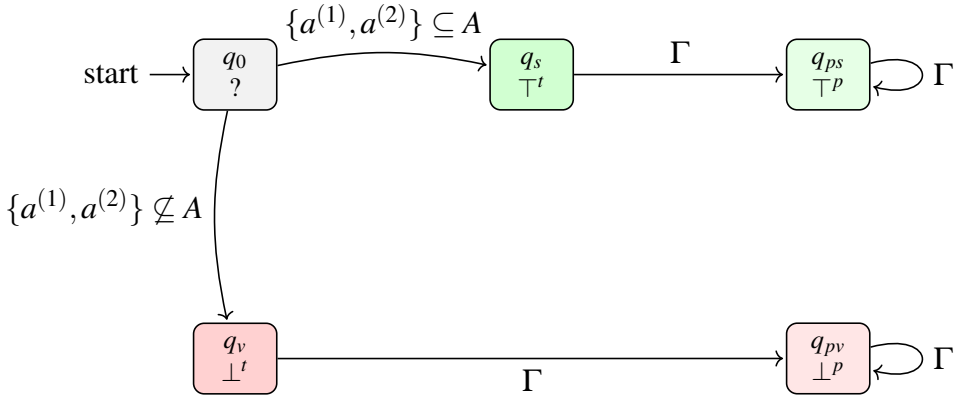


Figure 4.10: Compact 5-verdict Moore machine for the obligation literal $\mathbf{O}_1(a)$. Each node displays its internal state and the corresponding output verdict $\in \mathbb{V}_5$. The first joint execution of $a^{(1)}$ and $a^{(2)}$ yields $\top^t$, otherwise $\bot^t$; subsequent steps emit the post-frontier verdicts $\top^p$ or $\bot^p$.

The tight satisfaction monitor construction for the conjunction $C \wedge C'$, written $\mathsf{TSMC}_\wedge(C, C')$, is defined as:

$$\mathsf{TSMC}_\wedge(C, C') = (Q_\wedge, q_0^\wedge, \Gamma, \mathbb{V}_5, \delta_\wedge, \lambda_5^\wedge).$$

- *The state set is the Cartesian product*

  $$Q_\wedge = Q \times Q',$$

  with the initial state

  $$q_0^\wedge = (q_0, q_0').$$

- *The output function is*

  $$\lambda_5^\wedge(x, y) = \lambda_5^{comb}\big(\lambda_5(x),\ \lambda_5'(y)\big),$$

  where λ_5^{comb} is the conjunction-combination table:

$\lambda_5^{comb}(v_1, v_2)$	?	$\top^t$	$\bot^t$	$\top^p$	$\bot^p$
?	?	?	$\bot^t$	?	$\bot^p$
$\top^t$	?	$\top^t$	$\bot^t$	$\top^t$	$\bot^p$
$\bot^t$	$\bot^t$	$\bot^t$	$\bot^t$	$\bot^t$	$\bot^p$
$\top^p$	?	$\top^t$	$\bot^t$	$\top^p$	$\bot^p$
$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$

- *The transition function is the synchronous product:*

  $$\delta_\wedge\big((x, y), A\big) = \big(\delta(x, A),\ \delta'(y, A)\big),$$

  for all $(x, y) \in Q_\wedge$ and $A \in \Gamma$.

Partial dominance rules for $\wedge$. The monitor for $C \wedge C'$ checks both components at the same time and decides the global verdict according to the following rules:

- If any component gives $\bot^p$, the result is $\bot^p$:

  $$\forall v \in \mathbb{V}_5 : \quad \bot^p \sqcap v = \bot^p,$$

  That is, once a permanent violation appears, the whole conjunction is permanently violated.

- If any component gives $\bot^t$, and none is permanent, the result is $\bot^t$:

  $$\forall v \in \{?, \top^t, \top^p\} : \quad \bot^t \sqcap v = \bot^t,$$

  That is, a single tight violation makes the conjunction fail tightly.

- Tight and permanent success combine as the weakest success:

  $$\top^t \sqcap \top^t = \top^t, \qquad \top^t \sqcap \top^p = \top^t, \qquad \top^p \sqcap \top^p = \top^p,$$

  The conjunction is only permanently satisfied when both parts are permanent. *Note:* The case $\top^t \sqcap \top^p = \top^t$ reflects that if one component is exactly at its satisfaction frontier ($\top^t$) while the other is already past it ($\top^p$), the conjunction as a whole is determined by the later of the two, effectively placing the global frontier at the current step.

- If both sides are undecided or only partly satisfied, the result stays ?:

  $$? \sqcap v = ? \quad \text{for } v \in \{?, \top^t, \top^p\},$$

  The monitor waits until a clear outcome appears.

- The operator is symmetric:

  $$v_1 \sqcap v_2 = v_2 \sqcap v_1,$$

  The order of operands does not matter.

Lemma 36 (Correctness of the conjunctive monitor construction)**.** *Let*

$$\mathsf{TSMC}(C) = (Q, q_0, \Gamma, \mathbb{V}_5, \delta, \lambda_5) \quad \textit{and} \quad \mathsf{TSMC}(C') = (Q', q_0', \Gamma, \mathbb{V}_5, \delta', \lambda_5')$$

and let

$$\mathsf{TSMC}(C \wedge C') = (Q_\wedge, q_0^\wedge, \Gamma, \mathbb{V}_5, \delta_\wedge, \lambda_5^\wedge)$$

be the conjunction monitor defined in Definition 54. For every trace π, the output of the conjunction monitor satisfies:

$$\lambda_5^\wedge(\delta_\wedge(q_0^\wedge, \pi)) = [\![\pi \vDash C \wedge C']\!]_5 .$$

Proof sketch. The monitor $\mathsf{TSMC}_{\wedge}(\mathsf{TSMC}(C), \mathsf{TSMC}(C'))$ runs both component monitors in parallel and computes its output using the conjunction-combination table $\lambda_5^{\mathsf{comb}}$. The correctness follows from the prefix-based tight semantics of $C \wedge C'$.

- Permanent violation in either component produces $\bot^p$ immediately.
- A tight violation in one component produces $\bot^t$ whenever no permanent result is already present.
- Satisfaction requires both components to reach satisfaction states. If one is in $\top^p$ or $\top^t$ and the other is non-violating, the combined verdict matches the corresponding entry in the table.
- If both components remain undecided, the output is ?.

These cases match exactly the clauses for tight satisfaction, tight violation, post-satisfaction, and post-violation for the contract $C \wedge C'$. The monitor, therefore, correctly implements the tight semantics of conjunction. □

Sequential composition is the second binary operator of TACNL. Given two already constructed monitors $\mathsf{TSMC}(C)$ and $\mathsf{TSMC}(C')$, the monitor for the composite contract $C\,;C'$ must first execute C on the incoming trace and, once C reaches tight satisfaction, must continue execution with C' on the remaining suffix. The construction below reuses the state spaces of both components by redirecting transitions that correspond to the tight success of C into the initial state of C'. This yields a tight prefix monitor that exactly matches the semantics of sequential composition.

Definition 55 (Sequential Monitor Construction)**.** *Let*

$$\mathsf{TSMC}(C) = (Q, q_0, \Gamma, \mathbb{V}_5, \delta, \lambda_5) \quad \textit{and} \quad \mathsf{TSMC}(C') = (Q', q_0', \Gamma, \mathbb{V}_5, \delta', \lambda_5')$$

be the tight satisfaction monitors of C and C′. The tight satisfaction monitor for the sequential composition C ; C′, written $\mathsf{TSMC}_{;}(C, C')$*, is defined as:*

$$\mathsf{TSMC}_{;}(\mathsf{TSMC}(C), \mathsf{TSMC}(C')) = (Q_{;}, q_0^{;}, \Gamma, \mathbb{V}_5, \delta_{;}, \lambda_5^{;}).$$

- *The state set is*

  $$Q_{;} = (Q \setminus Q^{+}) \cup Q',$$

  where

  $$Q^{+} = \{\, x \in Q \mid \lambda_5(x) \in \{\top^t, \top^p\} \,\},$$

  and the initial state is

  $$q_0^{;} = q_0.$$

- *The transition function is*

$$\delta_{;}(q,A) = \begin{cases} q'_0 & \text{if } q \in Q \text{ and } \lambda_5(\delta(q,A)) = \top^t, \\ \delta(x,A) & \text{if } q \in Q \text{ and } \lambda_5(\delta(q,A)) \notin \{\top^t, \top^p\}, \\ \delta'(x,A) & \text{if } q \in Q'. \end{cases}$$

- *The output function is*

$$\lambda^{;}_5(x) = \begin{cases} \lambda_5(x) & \text{if } x \in Q, \\ \lambda'_5(x) & \text{if } x \in Q'. \end{cases}$$

Intuition. The construction implements the idea that C must succeed tightly before C' becomes active. All states of C that already correspond to tight satisfaction ($\lambda_5(x) = \top^t$ or $\top^p$) are removed, since execution should never continue inside them. Any transition in C that would have entered such a removed state is redirected to the initial state q'_0 of C', thereby starting the second contract at the exact prefix where C achieves tight success. All other transitions behave exactly as in the original monitors. The resulting machine therefore behaves as C until C succeeds tightly, after which it behaves as C' for the remainder of the trace.

Lemma 37 (Correctness of the sequential monitor construction)**.** *Let* $\mathsf{TSMC}(C)$ *and* $\mathsf{TSMC}(C')$ *be the monitors for* C *and* C'*, and let* $\mathsf{TSMC}_{;}(\mathsf{TSMC}(C), \mathsf{TSMC}(C')) = (Q_{;}, q^{;}_0, \Gamma, \mathbb{V}_5, \delta_{;}, \lambda^{;}_5)$*. For every trace* π*, the monitor outputs the same verdict according to the sequence semantics:*

$$\lambda^{;}_5\big(\delta_{;}(q^{;}_0, \pi)\big) = [\![\pi \vDash C;C']\!]_5 .$$

Proof. The proof is by induction on the length of the input trace π.

Proof. Let $M := \mathsf{TSMC}_{;}(\mathsf{TSMC}(C), \mathsf{TSMC}(C')) = (Q_{;}, q^{;}_0, \Gamma, \mathbb{V}_5, \delta_{;}, \lambda^{;}_5)$. We prove by induction on $n := |\pi|$ that

$$\lambda^{;}_5\big(\delta_{;}(q^{;}_0, \pi)\big) \;=\; [\![\pi \vDash C;C']\!]_5 .$$

Base case ($n = 0$). For $\pi = \langle - \rangle$ we have $\delta_{;}(q^{;}_0, \langle - \rangle) = q_0$ and thus

$$\lambda^{;}_5\big(\delta_{;}(q^{;}_0, \langle - \rangle)\big) = \lambda_5(q_0) = ?.$$

This matches the tight semantics of $C;C'$ on the empty trace: no decisive prefix for C has been reached and C' cannot be active yet.

Inductive step. Assume the claim holds for all traces of length n. Let π be a trace of length $n+1$ and write $\pi = \pi' \circ \langle A \rangle$ with $|\pi'| = n$ and $A \in \Gamma$. Let $s := \delta_{;}(q_0^{;}, \pi')$ be the state reached after reading π'. We distinguish whether the monitor is already executing the right component.

Case 1: $s \in Q'$. Then, by Definition 55, $\delta_{;}(s,A) = \delta'(s,A)$ and $\lambda_5^{;}(s) = \lambda_5'(s)$, hence

$$\lambda_5^{;}\big(\delta_{;}(q_0^{;}, \pi)\big) = \lambda_5'\big(\delta'(s,A)\big).$$

Reaching a state of Q' means that the switch from C to C' has already happened at some earlier prefix, and from that point on the sequential monitor behaves exactly as the monitor for C' on the remaining suffix. Therefore the tight semantics of $C;C'$ on π is determined by the verdict of C' on that suffix. By correctness of $\mathsf{TSMC}(C')$ (established for the tight monitor construction), the right-hand side equals $[\![\pi \vDash C;C']\!]_5$.

Case 2: $s \in Q$. Let $t := \delta(s,A)$ be the next state of the left monitor. There are two subcases.

Case 2(a): $\lambda_5(t) = \top^t$. Then Definition 55 redirects the transition to q_0', so $\delta_{;}(s,A) = q_0'$ and

$$\lambda_5^{;}\big(\delta_{;}(q_0^{;}, \pi)\big) = \lambda_5^{;}(q_0') = \lambda_5'(q_0') = ?.$$

Semantically, this is exactly the step where C reaches its tight satisfaction frontier on the prefix π, and the contract C' becomes active on the remaining suffix. At this switching moment, the suffix is empty, so the status of C' is undecided, and the sequence verdict is ?. Thus the monitor output matches $[\![\pi \vDash C;C']\!]_5$.

Case 2(b): $\lambda_5(t) \notin \{\top^t, \top^p\}$. Then the switch has not occurred, so $\delta_{;}(s,A) = t$ and $\lambda_5^{;}(t) = \lambda_5(t)$. Therefore

$$\lambda_5^{;}\big(\delta_{;}(q_0^{;}, \pi)\big) = \lambda_5(t) = [\![\pi \vDash C]\!]_5 .$$

Before C reaches tight satisfaction, the sequence semantics is governed by the left component on the current prefix, hence $[\![\pi \vDash C]\!]_5 = [\![\pi \vDash C;C']\!]_5$ in this case. So the equality holds.

All cases agree with the semantic clauses of sequential composition, so the induction closes. □

Definition 56 (Reparation Monitor Construction)**.** *Let*

$$\mathsf{TSMC}(C) = (Q, q_0, \Gamma, \mathbb{V}_5, \delta, \lambda_5) \quad \textit{and} \quad \mathsf{TSMC}(C') = (Q', q_0', \Gamma, \mathbb{V}_5, \delta', \lambda_5')$$

be the tight satisfaction monitors for C and C'. The monitor for the reparation contract $C \blacktriangleright C'$, written

$$\mathsf{TSMC}_{\blacktriangleright}(C, C'),$$

is defined as the tuple

$$\mathsf{TSMC}_{\blacktriangleright}(\mathsf{TSMC}(C), \mathsf{TSMC}(C')) = (Q_{\blacktriangleright}, q_0^{\blacktriangleright}, \Gamma, \mathbb{V}_5, \delta_{\blacktriangleright}, \lambda_5^{\blacktriangleright}).$$

- *The state set is*

  $$Q_{\blacktriangleright} = (Q \setminus Q^{-}) \cup Q',$$

  where

  $$Q^{-} = \{\, q \in Q \mid \lambda_5(q) \in \{\bot^t, \bot^p\} \,\},$$

  and the initial state is

  $$q_0^{\blacktriangleright} = q_0.$$

- *The transition function is*

  $$\delta_{\blacktriangleright}(q, A) = \begin{cases} q'_0 & \text{if } q \in Q \text{ and } \lambda_5(\delta(q,A)) = \bot^t, \\ \delta(q,A) & \text{if } q \in Q \text{ and } \lambda_5(\delta(q,A)) \notin \{\bot^t, \bot^p\}, \\ \delta'(q,A) & \text{if } q \in Q'. \end{cases}$$

- *The output function is*

  $$\lambda_5^{\blacktriangleright}(q) = \begin{cases} \lambda_5(q) & \text{if } q \in Q, \\ \lambda'_5(q) & \text{if } q \in Q'. \end{cases}$$

Intuition. The reparation operator activates the secondary contract C' after the primary contract C reaches a tight failure. The construction mirrors the sequential case, except that the redirection applies to transitions leading to a tight violation of C rather than to those leading to tight satisfaction.

All states of C whose verdicts are already violating ($\lambda_5(q) \in \{\bot^t, \bot^p\}$) are removed. Every transition in C that would enter such a state is redirected to q'_0, the initial state of C'. This starts C' exactly at the first prefix where C tightly fails.

As long as no violation occurs, the monitor behaves exactly as C. Once a tight failure is detected, the remaining input is processed by C'.

Lemma 38 (Correctness of the reparation monitor construction)**.** *Let* $\mathsf{TSMC}(C)$ *and* $\mathsf{TSMC}(C')$ *be the monitors for C and C′, and let*

$$\mathsf{TSMC}_{\blacktriangleright}(\mathsf{TSMC}(C),\mathsf{TSMC}(C')) = (Q_{\blacktriangleright}, q_0^{\blacktriangleright}, \Gamma, \mathbb{V}_5, \delta_{\blacktriangleright}, \lambda_5^{\blacktriangleright})$$

be the monitor constructed in Definition 56. Then, for every trace π*, the monitor outputs the right verdict as specified by the tight semantics:*

$$\lambda_5^{\blacktriangleright}(\delta_{\blacktriangleright}(q_0^{\blacktriangleright}, \pi)) = [\![\pi \vDash C \blacktriangleright C']\!]_5 .$$

Proof sketch. The argument follows exactly the same scheme as the proof of Lemma 37 for sequential composition.

The reparation monitor behaves as the tight monitor for C as long as no tight violation occurs. By the correctness invariant of $\mathsf{TSMC}(C)$, all prefixes before the first tight violation are evaluated exactly as prescribed by the tight semantics of $C \blacktriangleright C'$.

At the unique prefix where C reaches a tight violation ($\bot^t$), the construction redirects the transition to the initial state q_0' of the monitor for C'. This corresponds precisely to the semantic clause that activates the reparation contract after the first decisive failure of C.

From that point on, the monitor evolves exactly as $\mathsf{TSMC}(C')$ on the remaining suffix. By the inductive correctness of the tight monitor construction for C', the emitted verdict coincides with the tight semantics of C' on that suffix.

Since all other transitions are unchanged and no ambiguity about the switching point is possible, the monitor output agrees with the tight semantics of $C \blacktriangleright C'$ on all finite traces. □

Remark 7 (Relation to Control Phases and Prior Work)**.** *The structural constructions of Definitions 55 and 56 can be seen as the static counterpart of the* control-phase *view used in runtime-verification frameworks. Instead of introducing an explicit control variable* $m \in \{\mathbf{L}, \mathbf{R}\}$*, with* $\mathbf{L}$ *for left-hand side and* $\mathbf{R}$ *for the right, to determine which component is active, our transformation achieves the same effect directly on the transition graph deleting the terminal states of the left monitor and redirecting the transitions that reach a decisive verdict (*$\top^t$ *for sequence,* $\bot^t$ *for reparation) to the initial state of the right monitor. This compile-time construction encodes the same operational behavior as a mode-augmented monitor that switches from* $\mathbf{L}$ *to* $\mathbf{R}$ *when the switching condition is met.*

*This idea parallels the phase-based runtime semantics proposed in the runtime verification literature [FFM09, BFFR18], where control modes are used to synchronize submonitors for sequential patterns such as "*after C succeeds, check C'*". In contrast, the reparation composition corresponds to the "*compensatory phase*" discussed in [GM06], where a secondary clause is activated after the primary obligation fails. Hence, the constructions in Definitions 55 and 56 realize at the automaton level the same phase shifts*

($\mathbf{L}\rightarrow\mathbf{R}$ *after tight success or tight failure) that those frameworks handle explicitly through control variables.*

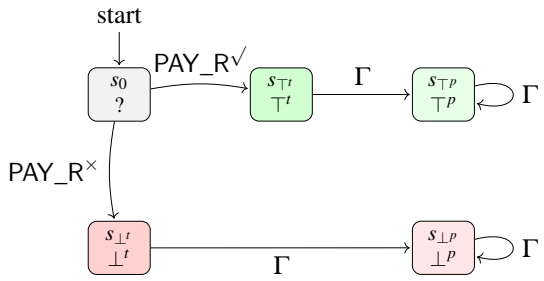
Example 57 (5-Output Moore Monitors for C_3 and $C_2 \wedge C_3$)**.** *The reparation contract* $C_3 = \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$ *combines two obligations: the primary duty* $\mathbf{O}_1(\mathsf{PAY\_R})$ *to pay rent, and the secondary reparation* $\mathbf{O}_1(\mathsf{PAY\_F})$ *to pay a late fee if the first is violated. In addition,* $C_2 = \mathbf{P}_1(\mathsf{OCC})$ *grants the tenant (agent 2) permission to occupy the property. Below, we show the literal monitors, their reparation composition, and the conjunction* $C_2 \wedge C_3$*, with verdict outputs* $\mathbb{V}_5 = \{?, \top^t, \bot^t, \top^p, \bot^p\}$.

We construct $C_2 \wedge C_3$ *by the synchronous product of the 5-output monitors for* $C_2 = \mathbf{P}_2(\mathsf{OCC})$ *and* $C_3 = \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$*, then keep only reachable states and minimize construction Figure 4.11. The Letter classes used on edges correspond to shorthand for literal satisfaction or violation conditions:*

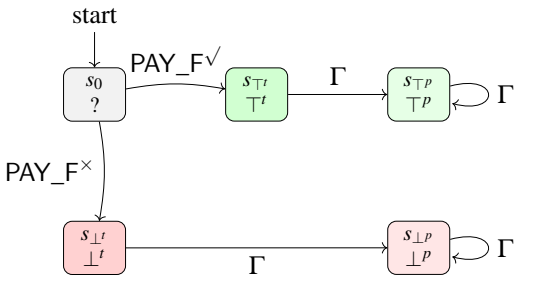
$$\mathsf{PAY\_R}^{\surd} := \{A \in \Gamma \mid \{\mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\} \subseteq A\}, \qquad \mathsf{PAY\_R}^{\times} := \Gamma \setminus \mathsf{PAY\_R}^{\surd},$$
$$\mathsf{PAY\_F}^{\surd} := \{A \in \Gamma \mid \{\mathsf{PAY\_F}^{(1)}, \mathsf{PAY\_F}^{(2)}\} \subseteq A\}, \qquad \mathsf{PAY\_F}^{\times} := \Gamma \setminus \mathsf{PAY\_F}^{\surd},$$
$$\mathsf{OCC}^{\surd} := \{A \in \Gamma \mid \mathsf{OCC}^{(2)} \in A \Rightarrow \mathsf{OCC}^{(1)} \in A\}, \qquad \mathsf{OCC}^{\times} := \Gamma \setminus \mathsf{OCC}^{\surd}.$$

Outputs follow the conjunction rule Λ*:* $\bot^t$ *if any conjunct tightly rejects,* $\bot^p$ *if any conjunct is post-reject,* $\top^t$ *when one conjunct hits* $\top^t$ *while the other is already at/past acceptance (*$\top^t$ *or* $\top^p$*),* $\top^p$ *if both are post-accept, and* ? *otherwise.*

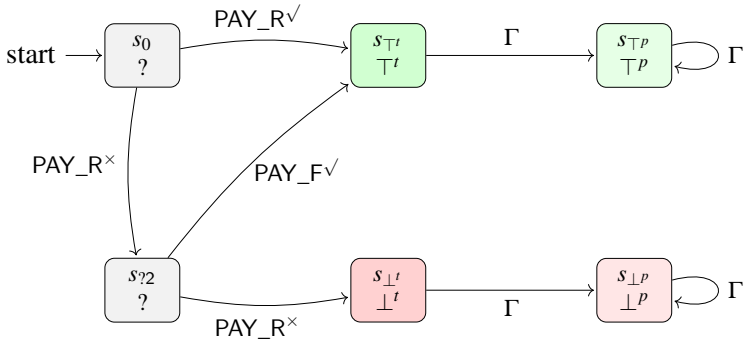
Reading 4.11d with the letter classes defined above: from s_0 *(pre) there are three behaviors: (i) If* $\mathsf{OCC}^{\surd} \wedge \mathsf{PAY\_R}^{\surd}$ *holds in the current letter, both conjuncts succeed (perm is respected, and the primary obligation is met), so the product emits* $\top^t$ *and moves to post-accept* $\top^p$*. (ii) If* $\mathsf{OCC}^{\times}$ *holds, the permission conjunct fails tightly, so the product emits* $\bot^t$ *and then* $\bot^p$ *forever. (iii) If* $\mathsf{OCC}^{\surd} \wedge \mathsf{PAY\_R}^{\times}$ *holds, the reparation branch of* C_3 *is activated, and we move to the waiting state* s_1 *(still ?). From* s_1*,* $\mathsf{PAY\_F}^{\surd}$ *discharges the reparation and triggers* $\top^t$*; otherwise* $\mathsf{PAY\_F}^{\times}$ *yields* $\bot^t$*. After* $\top^t$ *(accepting frontier), all continuations are in* $\top^p$*; after* $\bot^t$ *(reject frontier) all continuations are in* $\bot^p$*. Thus, the decisive index for the conjunction is the latter of the two successes when both succeed, or the earlier tight failure when any conjunct fails, exactly as the figure shows.*



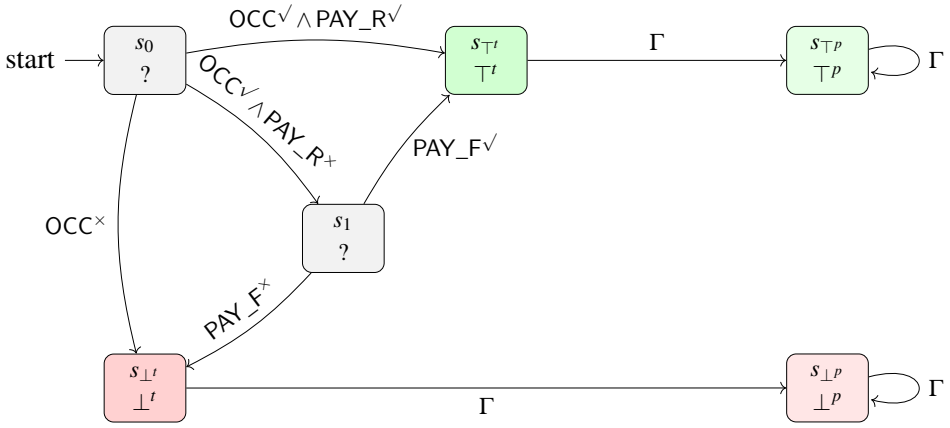
(a) $\mathsf{TSMC}_{\ell}(\mathbf{O}_1(\mathsf{PAY\_R}))$ Satisfaction occurs when both agents perform PAY_R.



(b) $\mathsf{TSMC}_{\ell}(\mathbf{O}_1(\mathsf{PAY\_F}))$ Satisfaction occurs when both agents perform PAY_F.



(c) Reparation composition $\mathsf{TSMC}_{\blacktriangleright}(\mathbf{O}_1(\mathsf{PAY\_R}),\mathbf{O}_1(\mathsf{PAY\_F}))$. The monitor activates PAY_F after tight failure of PAY_R.



(d) Conjunctive composition $\mathsf{TSMC}_{\wedge}(C_2,C_3)$, where $C_2 = \mathbf{P}_1(\mathsf{OCC})$.

Figure 4.11: Literal and composite 5-output Moore monitors for $C_3 = \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$. (a) and (b) monitors for Literal composing C_3 side by side; (c) Reparation composition C_3; (d) Conjunctive composition $C_2 \wedge C_3$, where $C_2 = \mathbf{P}_1(\mathsf{OCC})$ represents the tenant's power to occupy the property. $\mathsf{OCC}^{\surd} := \{A \mid \{\mathsf{OCC}^1\} \not\subset A \text{ or } \{\mathsf{OCC}^1,\mathsf{OCC}^1\} \subseteq A\}$. $\mathsf{PAY\_R}^{\surd} := \{A \mid \mathsf{PAY\_R}^{(1)},\mathsf{PAY\_R}^{(2)}\} \subseteq A$. The case $\mathsf{PAY\_F}^{\surd}$ is similarly defined as for $\mathsf{PAY\_R}^{\surd}$.

Construction for Binary Regular Expression-Contracts

Triggered contracts activate their body C once the triggering pattern *re* reaches its first tight match. Before that point, the monitor behaves exactly as the regular-expression monitor for *re* and emits only ?. Once the trigger fires, the monitor switches permanently to the contract monitor $\mathsf{TSMC}(C)$. If the pattern becomes impossible before it fires, the contract is vacuously satisfied. The construction follows the same blueprint as sequence and reparation, but applied to the decisive states of the regular expression monitor.

Definition 57 (Triggered Monitor Construction)**.**

$$\textit{Let } \mathsf{TSMC}(\mathsf{re}) = (Q_r, r_0, \Gamma, \mathbb{V}_5, \delta_r, \lambda_5^r) \quad \textit{and} \quad \mathsf{TSMC}(C) = (Q_c, c_0, \Gamma, \mathbb{V}_5, \delta_c, \lambda_5^c)$$

be five-valued tight satisfaction monitors for the regular expression re *and the contract C. The triggered monitor for* $\langle\langle re \rangle\rangle C$ *is the machine*

$$\mathsf{TSMC}_{trig}(\mathsf{TSMC}_{re}(\mathsf{re}), \mathsf{TSMC}(C)) := (Q_{trig}, q_0^{trig}, \Gamma, \mathbb{V}_5, \delta_{trig}, \lambda_5^{trig}).$$

- *The state space includes the open states of the trigger, the states of the contract, and two fresh sink states for vacuous success:*

  $$Q_{trig} := Q_r^{open} \cup Q_c \cup \{q_{vac}^{\top^t}, q_{vac}^{\top^p}\},$$

  where the reduced regular expressions states are:

  $$Q_r^{open} := \{\, q \in Q_r \mid \lambda_5^r(q) \in \{?, \top^t, \top^p\} \,\}.$$

  The initial state is

  $$q_0^{trig} := r_0.$$

- *The transition function* δ_{trig} *is defined in three parts:*

  1. ***Guard-active region*** $(q \in Q_r^{open})$***.*** *Let* $q' = \delta_r(q, A)$.

  $$\delta_{trig}(q,A) = \begin{cases} c_0 & \text{if } q' \in Q_r^{\top^t} \quad \textit{(first tight match: activate } C\textit{)}, \\ q_{vac}^{\top^t} & \text{if } q' \in Q_r^{\bot^t} \quad \textit{(trigger impossible: vacuous success)}, \\ q' & \text{if } q' \in Q_r^{?} \quad \textit{(guard still open)}. \end{cases}$$

  2. ***Contract-active region*** $(y \in Q_c)$***.***

  $$\delta_{trig}(y,A) := \delta_c(y,A).$$

3. ***Vacuous success sinks.***

$$\delta_{trig}(q_{vac}^{\top^t}, A) := q_{vac}^{\top^p}, \qquad \delta_{trig}(q_{vac}^{\top^p}, A) := q_{vac}^{\top^p}.$$

- *The output function is*

$$\lambda_5^{trig}(q) = \begin{cases} ? & \text{if } q \in Q_r^{open}, \\ \lambda_5^c(q) & \text{if } q \in Q_c, \\ \top^t & \text{if } q = q_{vac}^{\top^t}, \\ \top^p & \text{if } q = q_{vac}^{\top^p}. \end{cases}$$

Intuition. The trigger monitor is obtained with the same redirection recipe used for sequence, but applied to the trigger. Keep only states that are still open in the trigger (outputs in $\{?, \top^t, \top^p\}$). Remove its decisive states. Redirect every transition that would be the first tight match of the triggering regular expression (the step that enters $\top^t$) to the initial state of the contract monitor C. From that point on, the global output is exactly the output of C on the suffix. Redirect every transition that would make the trigger impossible (enter $\bot^t$ or $\bot^p$) to the fresh vacuous-success sink $q_{vac}^{\top^t}$, which emits $\top^t$ and then permanently $\top^p$. While the guard remains open, only the guard component advances and the product emits ?, so no premature verdict appears. This realizes the prefix clauses: success either because the trigger never becomes true (vacuous satisfaction), or because it fires at the earliest index and the suffix satisfies C; violation only if the guard fires and the suffix violates C.

Lemma 39 (Correctness of the triggered monitor construction)**.**

Let $\mathsf{TSMC}_{trig}(\mathsf{TSMC}_{re}(\mathsf{re}), \mathsf{TSMC}(C)) = (Q_{trig}, \Gamma, \delta_{trig}, q_0^{trig}, \lambda_5^{trig})$

be the monitor constructed in Definition 57 for $\langle\langle re \rangle\rangle C$*. Then, for every trace* π*,*

$$\lambda_5^{trig}\big(\delta_{trig}(q_0^{trig}, \pi)\big) \;=\; [\![\pi \vDash \langle\langle re \rangle\rangle C]\!]_5 \,.$$

Proof sketch. The proof follows the same pattern as for sequence and reparation, by induction on the length of π.

Base case. For $\pi = \langle - \rangle$ the monitor is in $q_0^{\text{trig}} = r_0$, the initial state of the guard. The value

$$\lambda_5^{\text{trig}}\big(\delta_{\text{trig}}(q_0^{\text{trig}}, \langle - \rangle)\big) = \lambda_5^r(r_0) = ?$$

coincides with the tight semantics of $\text{trig}[re]C$ on the empty trace: no trigger has fired, and no violation has occurred.

Inductive step. Assume the invariant holds for all prefixes of length n. Consider a prefix of length $n+1$ and its last letter A. There are three regions:

- *Guard-active region ($x \in Q_r^{open}$).* By construction, $\delta_{\text{trig}}(x,A)$ is:
  - c_0, if $\delta_r(x,A) \in Q_r^{\top^t}$. This is exactly the case where *re* reaches tight success for the first time. The next state is the initial state of $\mathsf{TSMC}(C)$, so subsequent behavior matches the semantics of C on the suffix. This realizes the clause "trigger fires at the earliest index and the suffix must satisfy C".
  - $q_{vac}^{\top^t}$, if $\delta_r(x,A) \in Q_r^{\bot^t}$, that is, the pattern becomes impossible. This state outputs $\top^t$ and transitions to the permanent success state $q_{vac}^{\top^p}$. This matches the semantics where the trigger never fires and $\langle\langle re \rangle\rangle C$ holds vacuously (tight success followed by post-success).
  - $\delta_r(x,A)$ if $\delta_r(x,A) \in Q_r^{?}$, in which case the guard remains open, and the global verdict stays ?. This matches the semantic clause that no decisive information is available as long as neither a match nor an impossibility has been detected.

  The inductive hypothesis on the guard monitor ensures that the moment of redirection coincides with the earliest decisive prefix of *re*.
- *Contract-active region ($y \in Q_c$).* Once the monitor has been redirected into c_0, all transitions are given by δ_c, and outputs by λ_5^c. The induction hypothesis for $\mathsf{TSMC}(C)$ gives

  $$\lambda_5^{\text{trig}}\big(\delta_{\text{trig}}(q_0^{\text{trig}}, \pi)\big) = \lambda_5^c\big(\delta_c(c_0, \pi')\big) = [\![\pi' \vDash C]\!]_5\,,$$

  where π' is the suffix after the trigger point. This matches the semantics of $\text{trig}[re]C$ on all traces where the trigger has fired.
- *Vacuous success region.* If the monitor enters $q_{vac}^{\top^t}$ or $q_{vac}^{\top^p}$, it correctly emits the success verdicts corresponding to a triggered contract whose trigger can no longer be satisfied.

Violation and satisfaction cases. If the trigger fires at some earliest index k and the suffix π^{k+1} violates C tightly or post, the monitor is in the contract-active region and outputs the corresponding $\bot^t$ or $\bot^p$, which is exactly $[\![\pi \vDash \text{trig}[re]C]\!]_5$ in this case. If the trigger never fires and the guard becomes impossible, the monitor outputs $\top^t$ and then $\top^p$, which matches vacuous satisfaction. In all other cases the output remains ?, as the semantics of $\text{trig}[re]C$ leaves the status undecided.

Conclusion. At each prefix, the monitor either simulates the guard with the correct decisive redirection points, simulates C on the correct suffix, or enters the correct vacuous success state. Hence for every trace π the monitor verdict $\lambda_5^{\text{trig}}(\delta_{\text{trig}}(q_0^{\text{trig}}, \pi))$ coincides with the five-valued tight semantics of $\text{trig}[re]C$. □

Definition 58 (Guarded Monitor Construction)**.**

Let $\mathsf{TSMC}(\mathsf{re}) = (Q_r, r_0, \Gamma, \mathbb{V}_5, \delta_r, \lambda_5^r)$ *and* $\mathsf{TSMC}(C) = (Q_c, c_0, \Gamma, \mathbb{V}_5, \delta_c, \lambda_5^c)$.

The guarded contract monitor for $\lceil re \rceil C$ *is the synchronous product*

$$\mathsf{TSMC}_{guard}(\mathsf{TSMC}_{re}(\mathsf{re}), \mathsf{TSMC}(C)) := (Q_r \times Q_c,\ (r_0, c_0),\ \Gamma,\ \mathbb{V}_5,\ \delta_{guard},\ \lambda_5^{guard}),$$

with

$$\delta_{guard}((x,y),A) := (\delta_r(x,A), \delta_c(y,A)).$$

The output verdict of the guarded monitor is determined by jointly inspecting the current verdicts of the guard and the contract components, according to the following case distinction.

$$\lambda_5^{guard}(x,y) = \begin{cases} \bot^t & \text{if } \lambda_5^r(x) \in \{?, \top^t, \top^p\} \text{ and } \lambda_5^c(y) = \bot^t, \\ \bot^p & \text{if } \lambda_5^r(x) \in \{?, \top^t, \top^p\} \text{ and } \lambda_5^c(y) = \bot^p, \\ \top^t & \text{if } \lambda_5^r(x) = \bot^t \text{ and } \lambda_5^c(y) \in \{?, \top^t, \top^p\}, \\ \top^p & \text{if } \lambda_5^r(x) = \bot^p \text{ and } \lambda_5^c(y) \in \{?, \top^t, \top^p\}, \\ \top^t & \text{if } \lambda_5^r(x) \in \{\top^t, \top^p\} \text{ and } \lambda_5^c(y) = \top^t, \\ \top^p & \text{if } \lambda_5^r(x) \in \{\top^t, \top^p\} \text{ and } \lambda_5^c(y) = \top^p, \\ ? & \text{otherwise.} \end{cases}$$

Intuition. The guard reads both monitors in lockstep and enforces:

- *Open guard.* While the guard is open $\big(\lambda_r \in \{?, \top^t, \top^p\}\big)$, i.e. exactly when $\pi \models_{Cl} re$, any tight or post failure of C becomes the global failure:

$$\pi \models_{\bot^t} \lceil re \rceil C \iff (\pi \models_{Cl} re) \wedge (\pi \models_{\bot^t} C),$$
$$\pi \models_{\bot^p} \lceil re \rceil C \iff (\pi \models_{Cl} re) \wedge (\pi \models_{\bot^p} C).$$

- *Guard impossible.* If the guard becomes impossible $\big(\lambda_r \in \{\bot^t, \bot^p\}\big)$, we accept provided C has not failed:

$$\pi \models_{\top^t} \lceil re \rceil C \iff (\pi \models_{\bot^t} re) \wedge (\pi \models_{Cl} C),$$
$$\pi \models_{\top^p} \lceil re \rceil C \iff (\pi \models_{\bot^t} re) \wedge (\pi \models_{\top^p} C).$$

- *Guard fired/closed.* When the guard has fired/closed $\big(\lambda_r \in \{\top^t, \top^p\}\big)$, we require C to (tight/post) succeed:

$$\pi \models_{\top^t} \lceil re \rceil C \iff (\pi \models_{Cl} re) \wedge (\pi \models_{\top^t} C),$$
$$\pi \models_{\top^p} \lceil re \rceil C \iff (\pi \models_{Cl} re) \wedge (\pi \models_{\top^p} C).$$

The resulting case table is symmetric and total, and it collapses to the expected two-valued clauses once tight/post outcomes are merged into satisfied vs. violated.

Lemma 40 (Correctness of the guarded monitor construction)**.**

$$\textit{Let } \mathsf{TSMC}_{guard}(\mathsf{TSMC}_{re}(\mathsf{re}), \mathsf{TSMC}(C)) = (Q_{guard}, q_0^{guard}, \Gamma, \mathbb{V}_5, \delta_{guard}, \lambda_5^{guard})$$

be the monitor constructed in Definition 58 for $\lceil re \rceil C$*. Then, for every finite trace* π*,*

$$\lambda_5^{guard}\big(\delta_{guard}(q_0^{guard}, \pi)\big) \;=\; [\![\pi \vDash \lceil re \rceil C]\!]_5 \,.$$

Proof sketch. The argument proceeds by induction on the length of π. The guarded monitor is a synchronous product of $\mathsf{TSMC}(re)$ and $\mathsf{TSMC}(C)$, with the output governed by the case distinction in Definition 58. Each region of the case table matches exactly one of the semantic clauses for $\lceil re \rceil C$.

Base case. For $\pi = \langle - \rangle$ we have

$$\lambda_5^{\text{guard}}\big(\delta_{\text{guard}}(q_0^{\text{guard}}, \langle - \rangle)\big) = \lambda_5^{\text{guard}}(r_0, c_0),$$

which yields ? in agreement with $[\![\langle - \rangle \vDash \lceil re \rceil C]\!]_5$.

Inductive step. Assume correctness for all prefixes of length n. Consider $\pi[n{+}1]$ with last letter A and let

$$(x', y') := \delta_{\text{guard}}((x,y), A) = (\delta_r(x,A), \delta_c(y,A)).$$

There are three semantic regions, corresponding to the three guard statuses.

- **Guard open** $\lambda_5^r(x) \in \{?, \top^t, \top^p\}$. This means π still possibly satisfies the guard. The guarded semantics requires that any tight or post failure of C becomes the global failure. The monitor table assigns $\bot^t$ or $\bot^p$ precisely in these cases, and ? otherwise, matching

  $$[\![\pi \vDash \lceil re \rceil C]\!]_5 = [\![\pi \vDash C]\!]_5 \quad \text{as long as } re \text{ is still open.}$$

- **Guard impossible** $\lambda_5^r(x) \in \{\bot^t, \bot^p\}$. This corresponds exactly to $\pi \models_{\bot^t} re$. The guarded semantics declares vacuous acceptance provided that C has not already failed. The output table assigns $\top^t$ or $\top^p$ if C has not failed, and propagates $\bot^t$ or $\bot^p$ if it has. This matches the semantic requirements for vacuous satisfaction.

- **Guard closed (triggered or concluded)** $\lambda_5^r(x) \in \{\top^t, \top^p\}$. In this region, the guard has fired or completed successfully, and the semantics require C to satisfy or to fail. The output table combines the post-accept and tight-accept verdicts of C with those of re exactly as demanded by the five-valued semantics:

$$[\![\pi \vDash \lceil re \rceil C]\!]_5 = [\![\pi \vDash C]\!]_5 \quad \text{once } re \text{ has closed.}$$

Conclusion. At each prefix of π, the guarded monitor outputs exactly the verdict prescribed by the five-valued semantics of $\lceil re \rceil C$. Thus

$$\lambda_5^{\text{guard}}\big(\delta_{\text{guard}}(q_0^{\text{guard}}, \pi)\big) = [\![\pi \vDash \lceil re \rceil C]\!]_5$$

for all traces π. □

Construction for repetition contracts

Repetition contracts describe behaviors that must be satisfied several times in sequence. The monitor construction follows the intuition that each repetition runs an independent copy of the monitor for C, with the next copy becoming active exactly when the previous one reaches tight success. For finite repetition C^n, this results in n chained monitors. For unbounded repetition $\mathbf{Rep}(C)$, we obtain an infinite cycle without ever reporting tight success.

The following constructions make these ideas explicit.

Definition 59 (Tight monitor construction for finite repetition $\text{frep}(n, C)$)**.**

Let $\mathsf{TSMC}(C) = (Q, q_0, \Gamma, \mathbb{V}_5, \delta, \lambda_5)$

be the five-valued tight satisfaction monitor for C. For $n \in \mathbb{N}^*$*, the tight monitor for the finite repetition contract* $frep(n, C)$ *is defined as*

$$\mathsf{TSMC}_{frep}(n, C) := (Q_{frep}, q_0^{frep}, \Gamma, \mathbb{V}_5, \delta_{frep}, \lambda_5^{frep}).$$

Disjoint copies. *For each* $i \in \{1, \ldots, n\}$*, create a disjoint copy of the base monitor:*

$$Q^{(i)} = \{q^{(i)} \mid q \in Q\}, \qquad \delta^{(i)}(q^{(i)}, A) = (\delta(q, A))^{(i)}, \qquad \lambda_5^{(i)}(q^{(i)}) = \lambda_5(q).$$

Write $q_0^{(i)}$ *for the copy of* q_0*.*

State space and start state.

$$Q_{frep} := \Big(\bigcup_{i=1}^{n} Q^{(i)}\Big) \cup \{q_{\top^p}, q_{\bot^p}\}, \qquad q_0^{frep} := q_0^{(1)},$$

where $q_{\top^p}$ and $q_{\bot^p}$ are fresh post-success and post-failure sinks.

Transition function. *For $1 \leq i \leq n-1$:*

$$\delta_{frep}(q^{(i)}, A) = \begin{cases} q_0^{(i+1)} & \text{if } \lambda_5^{(i)}(\delta^{(i)}(q^{(i)}, A)) = \top^t, \\ \delta^{(i)}(q^{(i)}, A) & \text{if } \lambda_5^{(i)}(\delta^{(i)}(q^{(i)}, A)) \notin \{\top^t, \top^p\}. \end{cases}$$

For the last copy $i = n$:

$$\delta_{frep}(q^{(n)}, A) = \begin{cases} q_{\top^p} & \text{if } \lambda_5^{(n)}(\delta^{(n)}(q^{(n)}, A)) = \top^t, \\ \delta^{(n)}(q^{(n)}, A) & \text{if } \lambda_5^{(n)}(\delta^{(n)}(q^{(n)}, A)) \notin \{\top^t, \top^p\}. \end{cases}$$

Sink states absorb:

$$\delta_{frep}(q_{\top^p}, A) = q_{\top^p}, \qquad \delta_{frep}(q_{\bot^p}, A) = q_{\bot^p}.$$

Output function. *For $q^{(i)} \in Q^{(i)}$:*

$$\lambda_5^{frep}(q^{(i)}) = \begin{cases} \bot^t & \text{if } \lambda_5^{(i)}(q^{(i)}) = \bot^t, \\ ? & \text{if } \lambda_5^{(i)}(q^{(i)}) \in \{?, \top^t, \top^p\}. \end{cases}$$

For sink states:

$$\lambda_5^{frep}(q_{\top^p}) = \top^p, \qquad \lambda_5^{frep}(q_{\bot^p}) = \bot^p.$$

This completes the construction of the five-valued monitor for $frep(n, C)$.

Lemma 41 (Correctness of the finite repetition monitor)**.** *For every finite trace π, contract C, and $\mathsf{TSMC}_{frep}(n, C) := (Q_{frep}, q_0^{frep}, \Gamma, \mathbb{V}_5, \delta_{frep}, \lambda_5^{frep})$, the following holds*

$$\lambda_5^{frep}\big(\delta_{frep}(q_0^{frep}, \pi)\big) = [\![\pi \vDash C^n]\!]_5 .$$

In the next sections, we use the definitions of the different languages in the semantics of TACNL, along with automata constructions and transformations, to enable automatic detection of violations and the attribution of blame for synchronous interactions over contracts in TACNL.

Definition 60 (Tight monitor construction for unbounded repetition $\mathbf{Rep}(C)$)**.** *Let*

$$\mathsf{TSMC}(C) = (Q, q_0, \Gamma, \mathbb{V}_5, \delta, \lambda_5)$$

be the tight satisfaction monitor for C. The monitor for the unbounded repetition $\mathbf{Rep}(C)$ *is defined as*

$$\mathsf{TSMC}_{Rep} := (Q_\omega, q_0, \Gamma, \mathbb{V}_5, \delta_\omega, \lambda_5^\omega)$$

where the construction removes all success states of C and redirects tight success back to the initial state.

State space.

$$Q_\omega := Q \setminus Q_\top, \qquad Q_\top := \{\, q \in Q \mid \lambda_5(q) \in \{\top^t, \top^p\} \,\}.$$

Transition function. *For every* $q \in Q_\omega$ *and* $A \in \Gamma$*:*

$$\delta_\omega(q,A) = \begin{cases} q_0 & \text{if } \lambda_5(\delta(q,A)) = \top^t \quad (\textit{restart next iteration after tight success}), \\ \delta(q,A) & \text{otherwise}. \end{cases}$$

Output function. *The monitor never declares satisfaction:*

$$\lambda_5^\omega(q) = \begin{cases} \bot^t & \text{if } \lambda_5(q) = \bot^t, \\ ? & \text{if } \lambda_5(q) \in \{?, \top^t, \top^p\}. \end{cases}$$

This yields the tight five-valued monitor for $\mathbf{Rep}(C)$.

Lemma 42 (Correctness of the unbounded repetition monitor)**.** *For every finite trace* π*, and* $\mathsf{TSMC}_{\mathbf{Rep}()}(C) := (Q_\omega, q_0, \Gamma, \mathbb{V}_5, \delta_\omega, \lambda_5^\omega)$ *, the following holds*

$$\lambda_5^\omega\big(\delta_\omega(q_0, \pi)\big) = [\![\pi \vDash \mathbf{Rep}(C)]\!]_5 .$$

The monitor never emits $\top^t$ nor $\top^p$, because tight satisfaction of $\mathbf{Rep}(C)$ is false. It emits $\bot^t$ (and then permanently $\bot^p$) exactly when some iteration of C violates tightly, i.e.

$$\exists n \in \mathbb{N} : \ \pi \models_{\bot^t} C^n.$$

Otherwise, it remains in the undecided verdict ?.

Example 58 (Open ended contract monitor construction)**.** *W*

Example 59 (Open ended contract monitor construction)**.** *We illustrate the constructions for the guarded open-ended contract* $\lceil re \rceil \mathbf{Rep}(C_3)$ *where* $re = *^+; \{\mathsf{Notif\_T}^{(1)}\}$ *and* $C_3 = \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$*. The construction proceeds in three steps: the unbounded-repetition monitor for* C_3*, the regular-expression monitor for re, and finally the guarded product. The resulting automata are shown in Figure 4.12.*

(a) $\mathsf{TSMC}(\mathbf{Rep}(C_3))$ ***(Figure 4.12a).*** *The monitor contains the states* $q_0/?$*,* $q_w/?$*,* $q_{\perp^t}/\perp^t$*, and* $q_{\perp^p}/\perp^p$*. The meaning of the transitions is as follows.*

A joint payment $\mathsf{PAY\_R}^{\surd}$ *keeps the monitor at* q_0*. A missed payment* $\mathsf{PAY\_R}^{\times}$ *moves to the waiting state* q_w*. From* q_w*, a successful late fee* $\mathsf{PAY\_F}^{\surd}$ *restarts the cycle by returning to* q_0*. A failed repair* $\mathsf{PAY\_F}^{\times}$ *produces a tight violation and moves to* $q_{\perp^t}$*, which then steps on any letter to the permanent sink* $q_{\perp^p}$*. The sink loops on all letters.*

This matches the construction of Definition 60: tight success restarts a new cycle, and tight failure leads to a permanent violation.

(b) $\mathsf{TSMC}_{re}(re)$ ***for*** $re = *^+; \{\mathsf{Notif\_T}^{(1)}\}$ ***(Figure 4.12b).*** *The monitor has states* $s_0/?$*,* $s_1/?$*,* $s^t/\top^t$*, and* $s^+/\top^p$*.*

The regular-expression part reads arbitrary letters: $s_0 \xrightarrow{*} s_1$*. While no termination notice is received, the machine remains in* s_1 *via* $\overline{T} = \Gamma \setminus T$*. A letter in* $T = \{A \mid \mathsf{Notif\_T}^{(1)} \in A\}$ *produces a tight match and moves to* s^t*. Any continuation moves to the post-acceptance state* s^+*, which loops on all letters.*

(c) Guarded product $\mathsf{TSMC}_{\boldsymbol{guard}}(\mathsf{TSMC}_{re}(re), \mathsf{TSMC}(\mathbf{Rep}(C_3)))$ ***(Figure 4.12c).*** *The product is organised by verdict class: violating states on the left, undecided states in the centre, and accepting states on the right.*

Undecided region (centre). *The reachable combinations while the guard is open are* $s_0 \times q_0$*,* $s_1 \times q_w$*, and* $s_1 \times q_0$*. Their transitions follow the product rule* $(x,y) \xrightarrow{A} (\delta_r(x,A), \delta_c(y,A))$*. Examples include:*

$$
\begin{aligned}
&s_0 \times q_0 \xrightarrow{\mathsf{PAY\_R}^{\times}} s_1 \times q_w,\\
&s_1 \times q_w \xrightarrow{\mathsf{PAY\_F}^{\times}} (s_0, s_1) \times q_{\perp^t},\\
&s_1 \times q_w \xrightarrow{\mathsf{PAY\_F}^{\surd} \wedge \overline{T}} s_1 \times q_0,\\
&s_1 \times q_0 \xrightarrow{\mathsf{PAY\_R}^{\times} \wedge \overline{T}} s_1 \times q_w,\\
&s_0 \times q_0 \xrightarrow{\mathsf{PAY\_R}^{\surd}} s_1 \times q_0.
\end{aligned}
$$

Violation region (left). *Once the contract component reaches* $q_{\perp^t}$*, the guard is still open so the product outputs* $\perp^t$ *and moves on any letter to* $S \times q_{\perp^p}$*, which loops on every letter.*

Acceptance region (right). *When the guard fires on a letter in T, the product moves to $s^t \times (q_0, q_w)$ with verdict $\top^t$ provided the repetition contract has not violated. From there, any continuation leads to the post-acceptance state $s^+ \times Q$ that loops on all letters and emits $\top^p$.*

This behavior follows directly from Definition 58: while $\lambda_r \in \{?, \top^t, \top^p\}$ the guard is open, so any tight or post violation of $\mathbf{Rep}(C_3)$ becomes a violation of the guarded contract. Once the guard becomes true on T, the contract must satisfy $\mathbf{Rep}(C_3)$ from that point on, for the global verdict to be tight or post acceptance.

The guarded open-ended contract illustrates how the automaton constructions combine. The unbounded repetition of C_3 enforces an indefinite sequence of payments with repair, and the regular expression *re* recognizes the first occurrence of a termination notice. The guarded product synchronizes both behaviors: while the guard is open, all failures of $\mathbf{Rep}(C_3)$ propagate to the global verdict; once the guard closes, the open-ended obligation is discharged, and the monitor collapses to post acceptance provided that no violation has occurred. This example shows that the tight constructions scale to nested patterns of sequencing, repetition, guarding, and reparation while still preserving a direct correspondence with the prefix semantics of TACNL.

4.6.3 Forward-Looking Blame Semantics

The tight semantics of TACNL identify when a contract is satisfied or violated, but do not explain *who* caused a violation. To attribute responsibility, we refine the violation verdicts based on which party failed to meet the relevant normative requirement. We break down tight and post violations into those caused by agent 1, those caused by agent 2, those caused jointly by both, and those where neither party is responsible (blameless cases). Formally, we introduce tight violation verdicts

$$\bot^t_{\{1\}}, \quad \bot^t_{\{2\}}, \quad \bot^t_{\{1,2\}}, \quad \bot^t_{\emptyset},$$

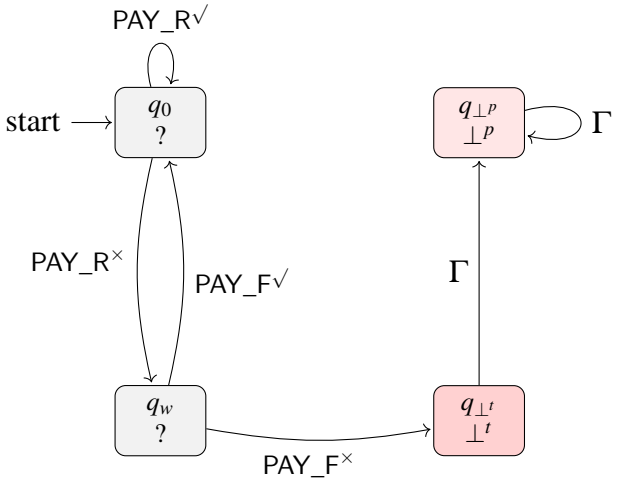
and their post-violation counterparts

$$\bot^p_{\{1\}}, \quad \bot^p_{\{2\}}, \quad \bot^p_{\{1,2\}}, \quad \bot^p_{\emptyset}.$$

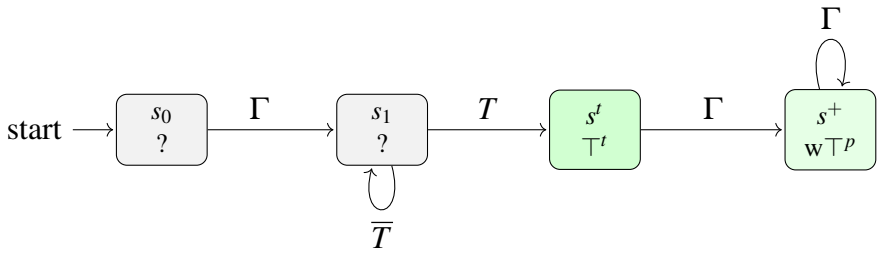
By replacing the undifferentiated violation verdicts of the five-valued semantics with these responsibility-aware variants, and keeping the three non-violating verdicts $?, \top^t, \top^p$, we obtain the forward-looking blame eleven-valued semantics

$$\mathbb{V}_{11} = \{?, \top^t, \top^p\} \cup \{\bot^t_S, \bot^p_S \mid S \subseteq \{1,2\}\}.$$

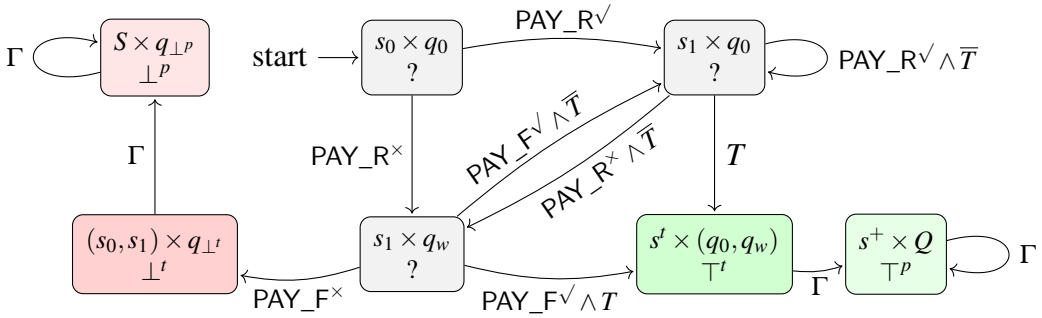
This refined judgement structure allows the monitors constructed in the next section to pinpoint the agents responsible for each contractual breach.



(a) Monitor for $\mathbf{Rep}(C_3)$.



(b) Monitor for the regular expression $\Gamma^+ \cdot \{terminate^{(1)}\}$.



(c) Tight satisfaction monitor construction for $\lceil \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)} \rceil \mathbf{Rep}(C_3)$.

Figure 4.12: Progressive construction for $\lceil (\Gamma^+ \cdot \mathsf{Notif\_T}^{(1)}) \rceil \mathbf{Rep}(C_3)$. Figure (c) is obtained by applying $\mathsf{TSMC}_{\text{guard}}$ to Figures (b) and (a). Here, roughly speaking, $\mathsf{PAY\_R}^{\surd} := \{A \in \Gamma \mid \{\mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\} \subseteq A\}$ denotes a joint payment event,$\mathsf{PAY\_R}^{\times} := \Gamma \setminus \mathsf{PAY\_R}^{\surd}$ denotes failure of payement. $T := \{A \in \Gamma \mid \mathsf{Notif\_T}^{(1)} \in A\}$ denotes the occurrence of a termination notice by agent 1, and $\overline{T} := \Gamma \setminus T$ denotes its absence.

Blame Rules for Literals

Definition 61 (Blame assignment for literals)**.** *Let $p \in \{1,2\}$ be the main subject of the norm and let $\overline{p}$ denote the other party. Let $\langle A \rangle$ be a single-step word with $A \in \Gamma$. We write $a^{(i)} \in A$ when party i attempts a in this step.*

Obligation $\mathbf{O}_p(a)$. *Violation occurs if and only if the joint execution does not happen. Blame principle: if the subject does not attempt, blame the subject; otherwise, blame the other party for not cooperating:*

$$\langle A \rangle \models_{\bot^t_{\{p\}}} \mathbf{O}_p(a) \stackrel{\text{def}}{=} a^{(p)} \notin A,$$
$$\langle A \rangle \models_{\bot^t_{\{\overline{p}\}}} \mathbf{O}_p(a) \stackrel{\text{def}}{=} a^{(p)} \in A \wedge a^{(\overline{p})} \notin A.$$

These two cases partition a tight violation of $\mathbf{O}_p(a)$.

Prohibition $\mathbf{F}_p(a)$. *Violation requires the joint act to occur. Since the subject should refrain, blame the subject:*

$$\langle A \rangle \models_{\bot^t_{\{p\}}} \mathbf{F}_p(a) \stackrel{\text{def}}{=} \{a^{(1)}, a^{(2)}\} \subseteq A.$$

If only one agent attempts the action, it does not violate a prohibition, so no other possible blame arises. An agent cannot be blamed for a prohibition that was not assigned to it.

Power $\mathbf{P}_p(a)$. *Blame occurs only when the subject of the power attempts and the other party withholds cooperation, and the blame goes to the other party:*

$$\langle A \rangle \models_{\bot^t_{\{\overline{p}\}}} \mathbf{P}_p(a) \stackrel{\text{def}}{=} a^{(p)} \in A \wedge a^{(\overline{p})} \notin A.$$

Invalid ($\bot$) is blameless. $\neg(\langle A \rangle \models_{\bot^t_S} \top)$ *for all S.* $\langle A \rangle \models_{\bot^t_\emptyset} \bot$ *by convention (unsatisfiable literal with no party subject).*

Post violation (prefix closure of blame). *Tight blame persists to extensions, and post blame is exactly "some earlier tight blame":*

$$\langle A \rangle \models_{\bot^t_S} \ell \implies \forall \pi \neq \langle - \rangle : \langle A \rangle \circ \pi \models_{\bot^p_S} \ell$$

Remark 8 (No joint blame at the literal level)**.** *Each literal is decided on a single step, and the only responsibility split is between the subject and the other party: either the subject fails to attempt, or the other party fails to cooperate, or no violation occurs. Hence, for literals, the blame set is always a singleton $S \in \{\{1\},\{2\}\}$ (or empty for $\bot$), never $\{12\}$.*

Example 60 (Obligation, prohibition, and power blame)**.** *By fixing $p = 1$, $\overline{p} = 2$. Consider the following letters $A \in \Gamma$:*

Obligation $\mathbf{O}_1(a)$.

$A = \emptyset$:	$\langle A \rangle \models_{\bot^t_{\{1\}}} \mathbf{O}_1(a)$	*(subject of the obligation did not attempt)*
$A = \{a^{(2)}\}$:	$\langle A \rangle \models_{\bot^t_{\{1\}}} \mathbf{O}_1(a)$	*(subject did not attempt)*
$A = \{a^{(1)}\}$:	$\langle A \rangle \models_{\bot^t_{\{2\}}} \mathbf{O}_1(a)$	*(other party did not cooperate)*
$A = \{a^{(1)}, a^{(2)}\}$:	*no violation*	*(joint execution present).*

Prohibition $\mathbf{F}_1(a)$.

$A = \{a^{(1)}, a^{(2)}\}$:	$\langle A \rangle \models_{\bot^t_{\{1\}}} \mathbf{F}_1(a)$	*(subject should have refrained)*
$A = \{a^{(1)}\}$:	*no violation*	*The prohibited action was not successful*

Power $\mathbf{P}_1(a)$.

$A = \{a^{(1)}\}$:	$\langle A \rangle \models_{\bot^t_{\{2\}}} \mathbf{P}_1(a)$	*(subject asked, other party withheld)*
$A = \{a^{(1)}, a^{(2)}\}$:	*no violation*	*(properly supported)*
$A = \{a^{(2)}\}$:	*no violation*	*(no unsupported subject attempt).*

Blame Propagation in Contracts

Conjunction. For $S \subseteq \{1,2\}$ of agent(s), and two contract C and C' from TACNL and a synchronous trace π, blame is defined for the conjunction $C \wedge C'$ is defined as:

$$\pi \models_{\bot^t_S} (C \wedge C') \iff \begin{cases} \pi \models_{\bot^t_S} C \text{ and } \pi \models_{\perp\!\!\!\perp} C', \\ \pi \models_{\bot^t_S} C' \text{ and } \pi \models_{\perp\!\!\!\perp} C, \\ \pi \models_{\bot^t_{S_1}} C \text{ and } \pi \models_{\bot^t_{S_2}} C' \text{ with } S = S_1 \cup S_2. \end{cases}$$

Where $\perp\!\!\!\perp$ stand for a non violation verdict, i.e, $\perp\!\!\!\perp \in \{?, \top^p, \top^t\}$
Intuition. The three cases summarize the possible outcomes of forward-looking blame. The blame goes to the agent responsible for the first violation of the contract: so either C or C', but both contracts could be violated at the same time point, in this case, the agent or agents responsible for *both simultaneous* violation get the blame.

For the rest of the operators, blame follows a similar definition as the tight violation, with $k \in [0, |\pi|]$:

Sequence. For $S \subseteq \{1,2\}$, contracts C, C' in TACNL, and a synchronous trace π:

$$\pi \vDash_{\bot^t_S} (C;C') \iff \begin{cases} \pi \vDash_{\bot^t_S} C, \\ \exists \pi_k \vDash_{\top^t} C \text{ and } \pi^k \vDash_{\bot^t_S} C'. \end{cases}$$

Intuition. The first decisive failure before C has tightly succeeded belongs to C, so its blame propagates. Once C has tightly succeeded ($\top^t$) or is in post-success ($\top^p$), only C' can still fail, so the blame comes from C'. There is no tie, since C' becomes active only after C has tightly succeeded.

Reparation. For $S \subseteq \{1,2\}$, the blame for a reparation contract $C \blacktriangleright C'$ is defined as:

$$\pi \vDash_{\bot^t_S} (C \blacktriangleright C') \iff \exists k \text{ such that } \pi_k \vDash_{\bot^t} C \text{ and } \pi^k \vDash_{\bot^t_S} C'.$$

Intuition. A reparation clause becomes active only after a violation of C. The global blame set S, therefore, corresponds to the agents responsible for violating the reparation C' once it is triggered. The blame for C is not considered, as one cares only for the overall violation of the combined contracts.

Example 61 (Witness traces for all blame verdicts)**.** *We use* $\Sigma_C = \{\mathsf{PAY\_R}, \mathsf{PAY\_F}, \mathsf{OCC}\}$ *and letters* $A_t \subseteq \Gamma$ *with agent tags* $\cdot^{(1)}, \cdot^{(2)}$*. Recall*

$$C'_2 := \mathbf{P}_1(\mathsf{OCC}) \,;\, \mathbf{P}_1(\mathsf{OCC}), \qquad C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F}).$$

Tight blame for agent 1

$$\pi_1 = \langle A_0 \rangle, \quad A_0 = \{\mathsf{OCC}^{(1)}\}$$

Here, $\mathbf{P}_1(\mathsf{OCC})$ *and* $\mathbf{O}_1(\mathsf{PAY\_R})$ *are violated, the blame verdicts are:*

- *The tenant (1) gets blamed for violating the obligation to pay rent:* $\pi_1 \vDash_{\bot^t_{\{1\}}} \mathbf{O}_1(\mathsf{PAY\_R})$.
- *The landlord (2) gets blamed for violating the power of the tenant to occupy the flat:* $\pi_1 \vDash_{\bot^t_{\{2\}}} \mathbf{P}_1(\mathsf{OCC})$.

But the specification allows for the reparation $\mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$*. So consequently, no tight violation can be diagnosed at* $T = 1$*:*
$\pi_1 \models_? \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$*. Consequently, only the landlord gets the blame for the overall specification:*

$$\pi_1 \vDash_{\bot^t_{\{2\}}} C'_2 \wedge C_3.$$

Moreover, consider the trace of $\pi_2 := \langle\{\mathsf{OCC}^{(1)}\},\{\mathsf{OCC}^{(1)}\}\rangle$*, the extension of* π_1 *with the same event, as the blame is forward and tight looking, the blame is still assigned only to agent* 2 *(landlord) as they is responsible for the first violation.*

Let us consider instead the following trace $\pi_3 := \langle A'_0, A_1\rangle$ *with* $A'_0 := \{\mathsf{OCC}^{(1)},\mathsf{OCC}^{(2)}\}$ *and* $A_1 := \{\mathsf{OCC}^{(1)}\}$.

Here:

- *The landlord gets the blame at* $T = 2$ *for violating the power of the tenant to occupy the flat in the second month:*
  $\pi_3 \vDash_{\bot^t_{\{2\}}} \mathbf{P}_1(\mathsf{OCC})\ ;\ \mathbf{P}_1(\mathsf{OCC})$.
- *For the reparation clause* $\mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$ *we must distinguish two different situations in which the fine is not honored:*
  - *if the tenant never attempts to pay the fine, that is, no letter of the trace contains* $\mathsf{PAY\_F}^{(1)}$*, then the blame goes to agent 1:*
    $\pi \vDash_{\bot^t_{\{1\}}} \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$,
  - *if instead the tenant attempts to pay the fine and the landlord does not cooperate, for example, in a letter* A *with* $\mathsf{PAY\_F}^{(1)} \in A$ *and* $\mathsf{PAY\_F}^{(2)} \notin A$*, then the fine obligation is violated, and the blame goes to agent 2:*
    $\pi \vDash_{\bot^t_{\{2\}}} \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$.

Lemma 43 (Tight forward blame is deterministic)**.** *For every trace* $\pi \in \Gamma^*$ *and every contract* C *in* TACNL*, there exists a unique verdict* $v \in \mathbb{V}_{11}$ *such that* $\pi \vDash_v C$*. Equivalently, for all* $v, v' \in \mathbb{V}_{11}$*,*

$$\pi \vDash_v C \land \pi \vDash_{v'} C \implies v = v'.$$

Proof. By structural induction on C.

Literals. For $C = \ell$, the blame rules in Definition 14 distinguish violations by a finite case split on the current letter $A \in \Gamma$. In each case, at most one blame set is applicable. For instance, for $\mathbf{O}_p(a)$, either $a^{(p)} \notin A$ (yielding $\bot^t_{\{p\}}$), or $a^{(p)} \in A \land a^{(\bar{p})} \notin A$ (yielding $\bot^t_{\{\bar{p}\}}$), or the joint act occurs (no violation). The clauses for $\mathbf{F}_p(a)$ and $\mathbf{P}_p(a)$ are analogous. Post blame is uniquely determined because $\bot^p_S$ holds exactly when some earlier prefix had the corresponding tight blame $\bot^t_S$.

Boolean and temporal constructors. Assume determinism holds for strict subcontracts. For conjunction $C_1 \wedge C_2$, the inductive hypothesis yields unique verdicts v_1 and v_2 for C_1 and C_2 on π. The conjunction clause then produces a unique combined verdict: if exactly one side is a tight (or post) blame verdict, it is selected; if both are blame verdicts, the blame set is uniquely $S_1 \cup S_2$; otherwise the result is the unique non-violating combination.

For sequence $C_1;C_2$, the semantics activates C_2 only after C_1 reaches tight success ($\top^t$) and otherwise propagates the unique verdict of C_1. Thus there is no ambiguity between blaming C_1 and blaming C_2.

For reparation $C_1 \blacktriangleright C_2$, the repair is triggered by the first decisive violation of C_1. The triggering point is unique because the underlying tight regions are prefix-monotone, and the inductive hypothesis yields a unique verdict for the suffix evaluated against C_2. Hence the overall blame verdict is unique.

All remaining constructors follow the same pattern: control flow is determined by the underlying tight verdicts, and blame information is carried along deterministically by the inductive hypothesis. Therefore, for every π and C, exactly one verdict in $\mathbb{V}_{11}$ applies. □

Theorem 11 (Tight Forward Blame semantics is a refinement of tight forward satisfaction)**.** *The tight forward semantics is a refinement of forward tight semantics, that is for any contract C from* TACNL *and π over Γ^*, we have:*

$$[\![\pi, C]\!]_5 = \begin{cases} ? & \text{iff } [\![\pi, C]\!]_{11} = ?, \\ \top^t & \text{iff } [\![\pi, C]\!]_{11} = \top^t, \\ \top^p & \text{iff } [\![\pi, C]\!]_{11} = \top^p, \\ \bot^t & \text{iff } \exists!\, S \subseteq \{1,2\} : [\![\pi, C]\!]_{11} = \bot^t_S, \\ \bot^p & \text{iff } \exists!\, S \subseteq \{1,2\} : [\![\pi, C]\!]_{11} = \bot^p_S. \end{cases}$$

Proof. We prove the statement by structural induction on the contract C.

Base case: literals. Let $C = \ell$ be a literal, and let $\pi \in \Gamma^*$ be a finite trace.

Case $|\pi| = 0$. By Definition 47, we have $\langle - \rangle \models_? \ell$ for every literal ℓ. Hence the corresponding five-valued verdict is the undecided one, namely $[\![\langle - \rangle, \ell]\!]_5 = ?$. On the blame side, there is no letter to trigger any tight blame clause, and post blame is defined only from an earlier tight blame. Therefore $[\![\langle - \rangle, \ell]\!]_{11} = ?$. This establishes the three non-violating cases in the displayed refinement table for $\pi = \langle - \rangle$.

Case $|\pi| \geq 1$. Write $\pi = \langle A \rangle \circ \pi'$ with $A \in \Gamma$.

First, consider the five-valued semantics. By Definition 47, literals are decided on the first letter: exactly one of $\langle A \rangle \models_{\top^t} \ell$ or $\langle A \rangle \models_{\bot^t} \ell$ holds.

Now consider the blame semantics for literals (your Definition "Blame assignment for literals"). If $\langle A \rangle \models_{\top^t} \ell$, then no blame violation verdict applies at the first step, so $[\![\pi, \ell]\!]_{11} \in \{?, \top^t, \top^p\}$ and the collapse leaves it unchanged. Hence $[\![\pi, \ell]\!]_5 \in \{?, \top^t, \top^p\}$ matches the same non-violating region.

If instead $\langle A \rangle \models_{\bot^t} \ell$, then by your literal blame rules the violation is refined into a (unique) blame set $S \subseteq \{1,2\}$: for $\mathbf{O}_p(a)$ there are exactly the two exclusive cases $a^{(p)} \notin A$ and $a^{(p)} \in A \wedge a^{(\bar{p})} \notin A$, for $\mathbf{F}_p(a)$ there is the single case $\{a^{(1)}, a^{(2)}\} \subseteq A$, and for $\mathbf{P}_p(a)$ there is the single case $a^{(p)} \in A \wedge a^{(\bar{p})} \notin A$. Thus $[\![\pi, \ell]\!]_{11} = \bot^t_S$ at the decisive step and, by the post clause, $[\![\langle A \rangle \circ \pi', \ell]\!]_{11} = \bot^p_S$ for every non-empty suffix π'. Collapsing erases S, yielding $\bot^t$ at the tight step and $\bot^p$ afterwards, which is exactly the five-valued classification of a literal (tight violation at the first letter, then post violation). Hence the refinement statement holds for literals.

Inductive step. Assume the statement holds for all strict subcontracts of C (and for the required suffix evaluations in the operators that split the trace). We show it holds for C by considering the outermost constructor.

Conjunction $C = C_1 \wedge C_2$. By the inductive hypothesis, for each $i \in \{1,2\}$ the blame verdict on C_i collapses to the five-valued verdict on C_i. In the blame semantics, the only difference from the five-valued semantics is that whenever a violation occurs, it carries a blame set, and when both sides violate at the same decisive point, the blame sets are unioned. Erasing the blame set therefore yields exactly the same coarse outcome as in the five-valued semantics: non-violating combinations stay in $\{?, \top^t, \top^p\}$, tight blame collapses to $\bot^t$, and post blame collapses to $\bot^p$. Hence the equivalence holds for $C_1 \wedge C_2$.

Sequence $C = C_1; C_2$. Both semantics use the same control flow: C_2 is evaluated only after the unique split point where C_1 reaches tight success, and otherwise the verdict is inherited from C_1. The blame semantics again only refines violations by attaching a blame set. By the inductive hypothesis on C_1 and on C_2 evaluated on the suffix beyond the split, collapsing the blame verdict yields exactly the five-valued verdict for $C_1; C_2$.

Reparation $C = C_1 \blacktriangleright C_2$. Both semantics activate C_2 precisely at the first tight violation of C_1. The five-valued semantics records only whether the composite ends in $\bot^t$ or $\bot^p$ (or remains non-violating), while the blame semantics records the same region but decorates violation with a blame set. By the inductive hypothesis for C_1 and for C_2 on the triggered

suffix, erasing the blame set yields the same coarse verdict as in the five-valued semantics. Hence the equivalence holds for $C_1 \blacktriangleright C_2$.

Repetition and regex constructors. For C^n, $\mathbf{Rep}(C)$, $\langle\langle re \rangle\rangle C$, and $\lceil re \rceil C$, the blame clauses mirror the five-valued clauses and differ only by refining violation outcomes with blame sets (and propagating them to the post region). Applying the inductive hypothesis to the involved subcontracts (and to the required suffixes) and then erasing the blame sets yields the corresponding five-valued verdict in each case.

Thus all constructors preserve the refinement property. Therefore the statement holds for all contracts C. □

4.6.4 From Tight Contract Satisfaction Monitor to Tight Blame Monitor

In the previous subsection, we defined a denotational blame semantics that refines tight and post violations by assigning responsibility to one or both agents. We now show that this refinement can be realized operationally. Unlike the tight satisfaction monitor, which groups all violations into a single generic $\bot^t$ verdict, the blame monitor must distinguish between different causes of violation. Therefore, we cannot simply relabel the outputs of the existing monitor; we must construct a new monitor whose state space explicitly encodes these distinctions.

Definition 62 (Blame monitor)**.** *The* blame monitor*, written $\mathcal{M}_{11}$, is a Moore machine whose output alphabet is the eleven-valued blame verdict set $\mathbb{V}_{11}$. Formally,*

$$\mathcal{M}_{11} = (Q, q_0, \Gamma, \mathbb{V}_{11}, \delta, \lambda_{11}),$$

where:

1. *The output alphabet is*

$$\mathbb{V}_{11} = \{?, \top^t, \top^p\} \cup \{\bot^t_S, \bot^p_S \mid S \subseteq \{1,2\}\}.$$

2. *Q is the set of states and $q_0 \in Q$ is the initial state,*
3. *$\Gamma = 2^\Sigma$ is the input event alphabet,*
4. *$\delta : Q \times \Gamma \to Q$ is the transition function,*
5. *$\lambda_{11} : Q \to \mathbb{V}_{11}$ is the state output function.*

We now define the construction function $\mathsf{BMC}(C)$ inductively. This follows the same structural approach as the tight satisfaction monitor construction $\mathsf{TSMC}(C)$ (Definition 52), but creates distinct states for distinct blame assignments.

Definition 63 (Blame Monitor Construction)**.** *The* blame monitor construction *is a function* $\mathsf{BMC}(C)$ *defined inductively on the structure of the contract* C*.*

Base Case: Literals (ℓ). *For any literal ℓ (including obligations, permissions, and constants $\top, \bot$), the monitor is defined as:*

$$\mathsf{BMC}_{lit}(\ell) = (Q_\ell, q_0, \Gamma, \mathbb{V}_{11}, \delta_\ell, \lambda_{11}).$$

- ***State Space:*** *The set Q_ℓ contains a start state, success states, and specific blame states for any valid blame set S associated with ℓ (where $S = \emptyset$ for $\bot$):*

$$Q_\ell = \{q_0, q_s, q_{ps}\} \cup \bigcup_S \{q^S_{\bot^t}, q^S_{\bot^p}\}.$$

- ***Transitions:*** *The initial transition is determined directly by the denotational blame rules on the input letter A:*

$$\delta_\ell(q_0, A) = \begin{cases} q_s & \text{if } \langle A\rangle \models_{\top^t} \ell \quad (\text{always true for } \top), \\ q^S_{\bot^t} & \text{if } \langle A\rangle \vDash_{\bot^t_S} \ell. \end{cases}$$

  Subsequent transitions capture the irrevocability of the verdicts:

$$\delta_\ell(q_s, \Gamma) = q_{ps}, \quad \delta_\ell(q_{ps}, \Gamma) = q_{ps}, \quad \text{and} \quad \delta_\ell(q^S_{\bot^t}, \Gamma) = q^S_{\bot^p}, \quad \delta_\ell(q^S_{\bot^p}, \Gamma) = q^S_{\bot^p}.$$

- ***Outputs:*** *λ_{11} maps states to their corresponding verdicts:*
  $q_0 \mapsto ?$, $q_s \mapsto \top^t$, $q_{ps} \mapsto \top^p$, $q^S_{\bot^t} \mapsto \bot^t_S$, and $q^S_{\bot^p} \mapsto \bot^p_S$.

Inductive Step: Other Operators. *For Sequence ($C_1;C_2$), Reparation ($C_1 \blacktriangleright C_2$), Repetition ($frep(n,C)$, $\mathbf{Rep}(C)$), and Guarded contracts, the construction follows the exact same topological logic as the Tight Satisfaction Monitor constructions (Definitions 55 to 60), with one key adaptation:*

- ***Matching Blame States:*** *Wherever the tight monitor construction redirects a generic violation state (outputting $\bot^t$), the blame monitor construction redirects* all *corresponding specific blame states (outputting $\bot^t_S$ for any S).*
- ***Example (Reparation):*** *In $\mathsf{TSMC}_\blacktriangleright$, transitions to any state q where $\lambda_5(q) = \bot^t$ are redirected to the initial state of C_2. In $\mathsf{BMC}_\blacktriangleright$, transitions to* any *state q where $\lambda_{11}(q) = \bot^t_S$ (regardless of S) are redirected to the initial state of C_2.*

This preserves the control-flow semantics of the operators: for example, in reparation, any *fault by* any *party in the primary contract triggers the secondary contract, effectively masking the initial blame in favor of the reparation's outcome.*

Theorem 12 (Correctness of the Blame Monitor)**.** *Let C be a contract in* TACNL*. For every finite trace π, the output of the blame monitor* $\mathsf{BMC}(C)$ *equals the denotational blame verdict:*

$$\lambda_{11}\big(\delta(q_0,\pi)\big) = \mathsf{Blame}(C,\pi).$$

Proof. The proof proceeds by structural induction on C.

- **Literals:** The state splitting in BMC_{lit} explicitly encodes the blame partition rules. For $\mathbf{O}_p(a)$, the transition δ_ℓ ensures that a trace with $a^{(p)} \notin A$ reaches $q^{p}_{\perp^t}$, while a trace with blocked cooperation reaches $q^{\bar{p}}_{\perp^t}$. This matches the semantic definition.
- **Conjunction:** The product construction explores all pairs of states. The definition of $\lambda^{\wedge}_{11}$ explicitly calculates $S_1 \cup S_2$ when both components are in violation states, and selects the single responsible party when only one violates. This matches the semantic clause $\pi \vDash_{\perp^t_{S_1\cup S_2}} C_1 \wedge C_2$.
- **Other Operators:** The correctness relies on the fact that operators like Sequence and Reparation are defined by switching control based on the *presence* of a violation (or success), not the *content* of the blame. For instance, in $C_1 \blacktriangleright C_2$, the semantics state that if C_1 fails (regardless of S), we evaluate C_2. The monitor construction realizes this by redirecting all transitions targeting any $\perp^t_S$-labeled state of C_1 to the start of C_2. Thus, the initial blame S is discarded (masked) exactly as prescribed by the semantics, and the final verdict is determined by C_2.

□

Example 62 (Blame Monitor for $C_2 \wedge C_3$)**.** *Let us recall that $C_2 = \mathbf{P}_1(\mathsf{OCC})$ represents the tenant's power to occupy the property, and $C_3 = \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$ represents the obligation to pay rent, repaired by paying a fine. The following figure shows the blame refinement of the monitor in Figure 4.11d. The generic violation state is partitioned into specific blame verdicts based on the cause of the failure.*

Although the previous example is constructed using a conjunction, the reparation operator within C_3 delays the assignment of blame for the payment obligation. Specifically, if the tenant fails to pay rent, the monitor transitions to a waiting state for the repair (outputting ?) rather than immediately emitting a violation verdict. Consequently, it is impossible for both conjuncts to return a tight violation $\perp^t$ at the same initial step. To illustrate a scenario where the monitor can output the joint blame verdict $\perp^t_{12}$, we consider the reparation-free reduction of the specification: $\mathbf{P}_1(\mathsf{OCC}) \wedge \mathbf{O}_1(\mathsf{PAY\_R})$.

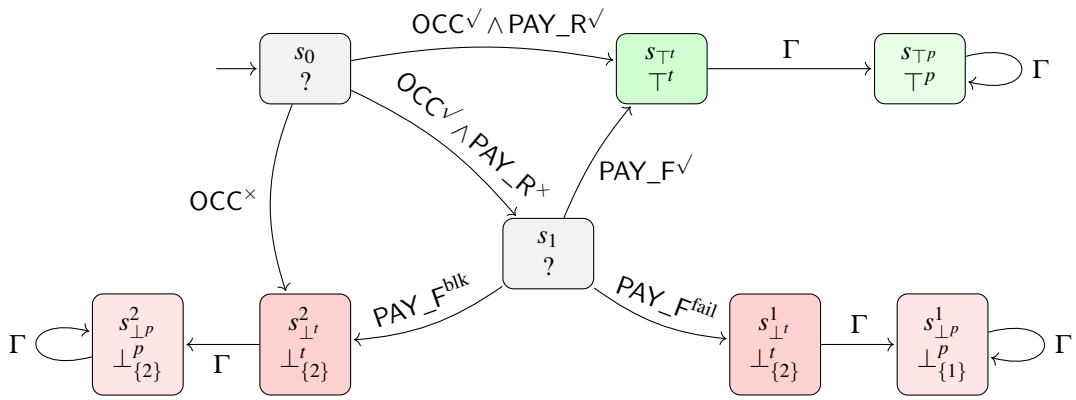


Figure 4.13: Blame Monitor $\mathcal{BM}(C_2 \wedge C_3)$. **Changes from Tight Monitor:** The state $s_{\perp^t}$ is split into $s^2_{\perp^t}$ (Landlord blame) and $s^1_{\perp^t}$ (Tenant blame). **Edge Definitions:** $\mathsf{OCC}^{\times}$: Tenant attempts OCC, Landlord blocks. $\mathsf{PAY\_F}^{\text{fail}}$: Tenant does not attempt PAY_F ($\mathsf{PAY\_F}^{(1)} \notin A$). $\mathsf{PAY\_F}^{\text{blk}}$: Tenant attempts PAY_F, Landlord blocks.

Example 63 (Blame Monitor with double blame)**.** *The following monitor in Figure 4.14 shows the emergence of joint blame. From the initial state s_0, three distinct violation paths are possible depending on who fails. The path to $s^{12}_{\perp^t}$ represents the simultaneous failure of both parties.*

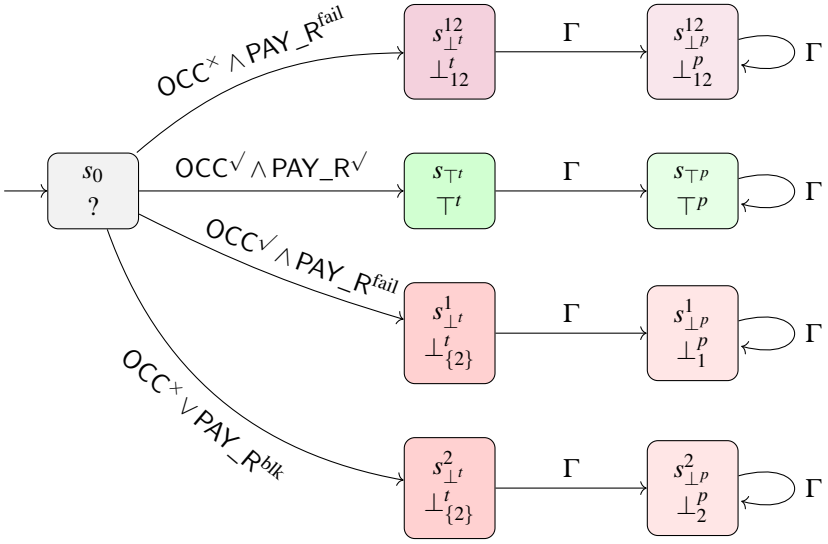


Figure 4.14: Blame Monitor for $\mathbf{P}_1(\mathsf{OCC}) \wedge \mathbf{O}_1(\mathsf{PAY\_R})$.
Edge Definitions:
$\mathsf{PAY\_R}^{\text{fail}}$: Tenant does not attempt payment ($\mathsf{PAY\_R}^{(1)} \notin A$).
$\mathsf{OCC}^{\times}$: Tenant attempts occupation, Landlord blocks.
$\mathsf{PAY\_R}^{\text{blk}}$: Tenant attempts to pay and Landlord blocks.
The state $s^{12}_{\perp^t}$ is reached only when both violations occur in the same step.

Example 64 (Blame Monitor for $\mathbf{Rep}(C_3)$)**.** *The figure below shows the blame monitor for the unbounded repetition of the rent-and-reparation contract. The generic violation state $q_{\perp^t}$ from the standard monitor is split into $s^1_{\perp^t}$ and $s^2_{\perp^t}$. Crucially, once the monitor transitions to a post-violation sink (e.g., $s^1_{\perp^p}$), it loops on any input Γ. This demonstrates the "first blame" limitation: if the tenant is blamed for missing a fine, the monitor will never blame the landlord for any future misconduct.*

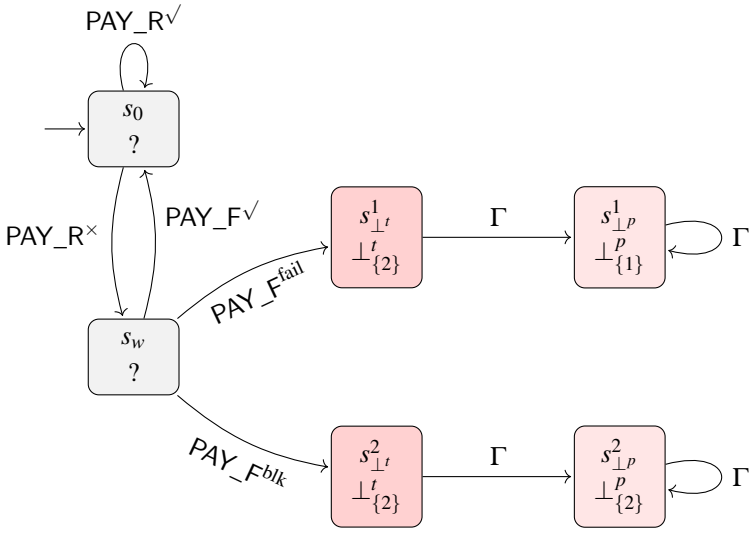


Figure 4.15: Blame Monitor for $\mathbf{Rep}(C_3)$. **Limitation:** If the trace reaches $s^1_{\perp^p}$ (Tenant blame), the monitor remains there forever. Even if the landlord subsequently blocks a valid payment attempt ($\mathsf{PAY\_F}^{\mathrm{blk}}$) in a future step, the verdict remains $\perp^p_1$.

4.6.5 Conclusion and Limitation

We have presented forward-looking semantics and a corresponding monitor construction that refine the standard satisfaction verdicts by assigning responsibility. This approach is computationally efficient and provides immediate feedback on the *status* of the contract, allowing for runtime enforcement and dispute resolution at the moment a breach occurs.

However, this prefix-based view naturally implies a limitation regarding the completeness of the violation history. The semantics is designed to detect the *first* decisive violation that renders the contract permanently unsatisfiable. Once the monitor transitions to a post-violation sink state ($\perp^p_S$), the verdict becomes immutable. A practical consequence of this property is that, even when processing infinite words or streams, the monitor can be programmed to halt execution immediately after the first tight violation is detected, as no future event can alter the blame assignment. Consequently, any subsequent violations committed by other agents at later time steps are effectively masked by the first failure.

This limitation is particularly evident in open-ended contracts involving the repetition operator. As illustrated by the monitor for $\mathbf{Rep}(C_3)$ (Figure 4.15), if the tenant is blamed for failing to pay the reparation in one cycle, the monitor enters the sink state $s^1_{\perp^p}$. Even if the interaction continues and the landlord subsequently violates their permission or obligation in a future cycle (e.g., by blocking a valid payment attempt), the monitor remains fixed on the initial verdict $\perp^p_1$. Therefore, while this framework is sufficient for determining *why* a contract failed, it does not support a cumulative accounting of *all* violations that occur throughout the lifespan of a long-running interaction. This is a notable constraint, as in law and normative systems, one typically has to account for all violations to determine the full extent of liability or penalties. Capturing such multi-point violations would require a mechanism to reset or parallelize monitoring threads after a failure, rather than absorbing them into a permanent sink.

Finally, extending this framework to support a cumulative blame semantics suggests interesting theoretical challenges. In particular, the interaction between timed operators and conjunctions complicates fault aggregation. For instance, determining whether overlapping failures in concurrent branches or repeated violations within sliding time windows should be counted as distinct or continuous breaches requires a more complex, possibly non-monotonic, judgement structure than the one presented here.

4.7 Quantitative Violation Semantics

The basic restriction of the forward-looking tight satisfaction semantics, as defined in the previous section, resides in its binary and prefix-closed nature. In that framework, the monitoring process halts definitively at the first tight violation. Despite being computationally efficient for stopping compliance systems (e.g., an access control system that is failing), this "first-fail" approach is insufficient for post-hoc auditing or dispute resolution. It masks the full history of non-compliance, failing to capture cumulative violations or distinct failures by multiple agents over time, a critical requirement for comprehensive legal responsibility. To tackle this limitation, it is necessary to transition from a boolean verdict to a *quantitative semantics*, changing the emphasis from the question "Did a violation occur?" to "How many violations occurred, and what was their magnitude?"

The Challenge of Temporal Scope. A prerequisite for counting violations is defining the temporal window over which the contract is evaluated. In contrast to traditional model checking, which commonly analyzes systems that are supposed to run forever, contract monitoring operates on finite, evolving prefixes. We identify two primary strategies for defining this scope:

1. **Static Pre-computation:** One might attempt to calculate a fixed duration T for a contract C. However, contracts containing unbounded repetitions ($\mathbf{Rep}(C)$) or input-dependent regular expressions (guards/triggers) do not have a statically determinable length.
2. **Dynamic On-the-Fly Detection:** The alternative is to determine the contract's status dynamically at every step. Here, the monitor continuously computes a *residual contract*—a formula representing what remains to be fulfilled given the history of events.

This work adopts the **Dynamic On-the-Fly** approach. This requires a structural tracking mechanism, which we formalize as the *Contract Progress Monitor* (*CPM*. The *CPM* cts as a derivative function: given a contract C and an incoming event, it computes the contract C' that must be enforced in the next time step. Isolating the state of the contract (via the *CPM* from the evaluation of compliance (via a scoring function) enables a modular framework that can track active duties, handle contrary-to-duty (CTD) transitions, and attribute blame with precision over time. The scoring function is critical in this framework, as it provides a quantitative evaluation of compliance. This approach not only determines whether obligations have been met, but also assesses the degree of compliance or non-compliance. The interaction with the *CPM* easures the magnitude of violations, presenting an in-depth and nuanced view of contractual adherence that improves accountability and enforcement precision.

Regarding the computational complexity of this framework, we claim that the set of reachable residual contracts is finite. The size of this set is determined by the size of the regular expressions and contract literals within the original contract C, which implies that the set can be computed in finite time. We aim to provide the formal proofs of this finiteness property and the algorithm for exhaustive computation as future work.

4.7.1 Contract Progress Monitor

The core of our measurement framework is the *progression function*, Prog. This function consumes the current single event and the current *state of a contract* to produce the *residual contract* for the subsequent step.

Definition 64 (Contract Progression Function)**.** *Let $\Gamma = 2^{\Sigma}$ be the event alphabet. We extend the set of contracts $\mathcal{C}$ with a distinguished symbol ε_C, representing a* discharged contract *(one that implies no further obligations).*

The progression function $\mathsf{Prog} : \Gamma^* \times \mathsf{TACNL} \to \mathsf{TACNL} \cup \{\varepsilon_C\}$ *is defined recursively. For the empty trace $\langle - \rangle$,* $\mathsf{Prog}(\langle - \rangle, C) := C$*. For a single event step $\langle A \rangle$ with $A \in \Gamma$, the function is defined on the structure of C as follows.*

Literals (State Update). *For any literal ℓ (obligation, permission, or prohibition):*

$$\mathsf{Prog}(\langle A\rangle, \ell) := \varepsilon_C$$

Rationale: *Structurally, a literal applies to a single time step. Once the A step occurs, the literal is consumed. Whether A satisfied or violated ℓ is immaterial to the* progression *(the duty is passed); the violation is recorded separately by the quantitative scoring function defined later.*

Conjunction (Parallel Progress).

$$\mathsf{Prog}(\langle A\rangle, C_1 \wedge C_2) := \mathsf{Prog}(\langle A\rangle, C_1) \wedge \mathsf{Prog}(\langle A\rangle, C_2)$$

We assume the symbolic identity where ε_C is the neutral element for conjunction: $\varepsilon_C \wedge C' \equiv C'$.

Sequence (Sequential Handover). *For a sequence $C_1; C_2$, progression determines if the current step concludes C_1:*

$$\mathsf{Prog}(\langle A\rangle, C_1; C_2) := \begin{cases} \mathsf{Prog}(\langle A\rangle, C_1); C_2 & \text{if } \mathsf{Prog}(\langle A\rangle, C_1) \neq \varepsilon_C, \\ \mathsf{Prog}(\langle A\rangle, C_2) & \text{if } \mathsf{Prog}(\langle A\rangle, C_1) = \varepsilon_C. \end{cases}$$

If C_1 is discharged by step A (i.e., its residual is ε_C), the monitor immediately activates the first step of the next portion C_2.

Reparation (Contrary-to-Duty Branching). *The reparation construct is unique because its structural progression depends on the satisfaction of the primary obligation. This is the only case where* Prog *relies on the tight satisfaction relation ($\models_{\top^t}, \models_{\bot^t}, \models_?$) to determine the path:*

$$\mathsf{Prog}(\langle A\rangle, C_1 \blacktriangleright C_2) := \begin{cases} \mathsf{Prog}(\langle A\rangle, C_1) \blacktriangleright C_2 & \text{if } \langle A\rangle \models_? C_1 \text{ (Pending)}, \\ \mathsf{Prog}(\langle A\rangle, C_2) & \text{if } \langle A\rangle \models_{\bot^t} C_1 \text{ (Repair)}, \\ \varepsilon_C & \text{if } \langle A\rangle \models_{\top^t} C_1 \text{ (Discharge)}. \end{cases}$$

If a violation occurs ($\langle A\rangle \models_{\bot^t} C_1$), the primary contract is discarded, and the secondary contract C_2 is activated immediately for the next *step.*

Repetition. *Repetition unrolls the contract one step at a time:*

$$\mathsf{Prog}(\langle A\rangle, \mathbf{Rep}(C)) := \mathsf{Prog}(\langle A\rangle, C); \mathbf{Rep}(C)$$

Guarded and Triggered Contracts. *These constructs rely on regular expression matching.*

$$\mathsf{Prog}(\langle A\rangle, \lceil re\rceil C) := \begin{cases} \varepsilon_C & \text{if } \langle A\rangle \models_{\top^t} \lceil re\rceil C \text{ (release)}, \\ \lceil \mathsf{Prog}_{re}(\langle A\rangle, re)\rceil \mathsf{Prog}(\langle A\rangle, C) & \text{otherwise (Guard persists)}. \end{cases}$$

$$\mathsf{Prog}(\langle A\rangle, \langle\!\langle re\rangle\!\rangle C) := \begin{cases} \varepsilon_C & \text{if } \langle A\rangle \models_{\bot^t} re \text{ (Trigger failed)}, \\ C & \text{if } \langle A\rangle \models_{\top^t} re \text{ (Trigger fires)}, \\ \langle\!\langle \mathsf{Prog}_{re}(\langle A\rangle, re)\rangle\!\rangle C & \text{if } \langle A\rangle \models_{?} re \text{ (Trigger pending)}. \end{cases}$$

Where the regular progress function $\mathsf{Prog}_{re} : 2^{\Gamma} \times \mathsf{TACNL}_{re} \to \mathsf{TACNL}_{re}$ *is defined only when* $\langle A\rangle \models_{?} \mathsf{re}$ *(where* $\langle A\rangle$ *is the trace consisting of the single event A). We do not need to use* ε_{re} *as we did for the contract as* ε *is already defined in the syntax of regular expression, with* $\mathsf{Prog}_{re}(\langle A\rangle, \varepsilon) := \varepsilon$.

Atomic regular expressions *For a set of actions* $A \in \Gamma$ *we have:* $\mathsf{Prog}_{re}(\langle A\rangle, A') := \varepsilon$.

Concatenation *For* $\mathsf{re}\cdot\mathsf{re}'$, *the progression is defined as:*

$$\mathsf{Prog}_{re}(\langle A\rangle, \mathsf{re}\cdot\mathsf{re}') := \mathsf{Prog}_{re}(\langle A\rangle, re)\cdot re'.$$

Union *For* $re \mid re'$, *the progression is defined by distinguishing cases based on violation:*

$$\mathsf{Prog}_{re}(\langle A\rangle, re \mid re') := \begin{cases} \mathsf{Prog}_{re}(\langle A\rangle, re') & \text{if } \langle A\rangle \models_{\bot^t} re, \\ \mathsf{Prog}_{re}(\langle A\rangle, re) & \text{if } \langle A\rangle \models_{\bot^t} re', \\ \mathsf{Prog}_{re}(\langle A\rangle, re) \mid \mathsf{Prog}_{re}(A, re') & \text{otherwise}. \end{cases}$$

n-repetition *For* re^n, *with* $n \in \mathbb{N}^*$ *the progression is defined as:*

$$\mathsf{Prog}_{re}(\langle A\rangle, re^n) := \mathsf{Prog}_{re}(\langle A\rangle, re)\cdot re^{n-1}.$$

+-Operator *For* re^+, *the progression is defined as:*

$$\mathsf{Prog}_{re}(\langle A\rangle, re^+) := \mathsf{Prog}_{re}(\langle A\rangle, re) \mid \mathsf{Prog}_{re}(\langle A\rangle, re)\cdot re^+.$$

Moving up for the special case of single event traces, for $\pi = \langle A\rangle \circ \pi'$ *with* $A \in \Gamma$ *the first event and* $\pi' \in \Gamma^*$ *the remaining suffix. Then:*

$$\mathsf{Prog}(\langle A\rangle \circ \pi', C) := \begin{cases} \varepsilon_C & \text{if } \mathsf{Prog}(\langle A\rangle, C) = \varepsilon_C, \\ \mathsf{Prog}\big(\pi', \mathsf{Prog}(\langle A\rangle, C)\big) & \text{otherwise}. \end{cases}$$

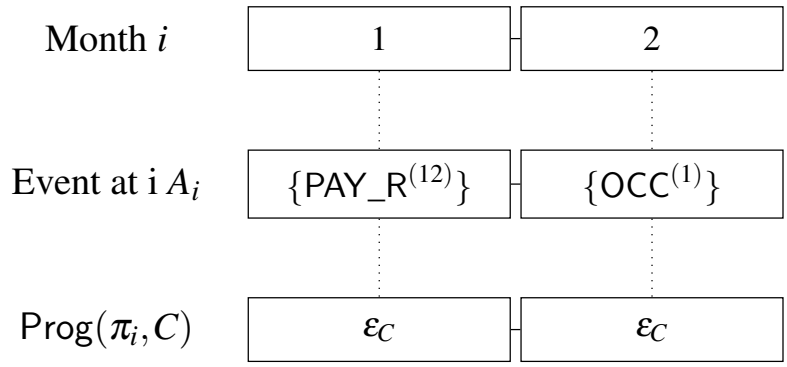


Figure 4.16: Progression on $\mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$ with a trace for which no reparation is required

The helper function Prog_{re} computes the residual regular expression following an idea from Brzozowski's derivative [Brz64] of the regular expression. In our case, it is tuned for the tight semantics.

Lemma 44 (Termination)**.** *For each finite prefix π, $\mathsf{Prog}(\pi, C)$ terminates in at most $\min|\pi|$ recursive steps and the worst case is considering the worst regular expression such as the guard of the use case.*

To illustrate the *CPM* the evolution of the residual contract under different traces is examined. The evaluation initiates with the fundamental building block of normative enforcement: the reparation.

Example 65 (Progression of Reparation)**.** *Consider the basic rental reparation clause:*

$$C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F}).$$

Scenario 1: Compliance. *In the first trace π, the tenant pays rent in Month 1 ($\mathsf{PAY\_R} \in A_1$). Since $\langle A_1 \rangle \models_{\top^t} \mathbf{O}_1(\mathsf{PAY\_R})$, the condition for discharge is met. The reparation structure collapses to ε_C, meaning no further obligations exist for Month 2.*

Scenario 2: Violation and Repair. *In trace π', the tenant fails to pay in Month 1 ($\mathsf{PAY\_R} \notin A'_1$). Here, $\langle A'_1 \rangle \models_{\bot^t} \mathbf{O}_1(\mathsf{PAY\_R})$. Consequently, the CPM ctivates the repair branch. The residual for Month 2 becomes $\mathbf{O}_1(\mathsf{PAY\_F})$, obliging the tenant to pay the fine.*

After establishing the evolution of single-step reparations, the analysis is extended to ongoing contracts. The following example demonstrates how the *CPM* andles infinite streams with recurring duties, showing how violations in one period persist into subsequent periods.

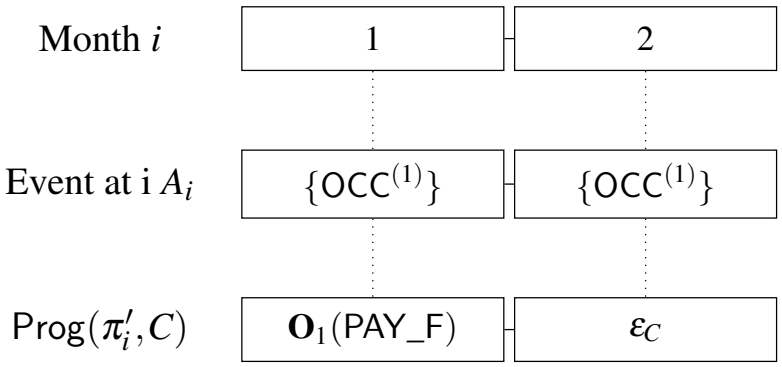


Figure 4.17: Progression on $\mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$ with a trace for which required a reparation

Example 66 (Progression of Infinite Repetition)**.** *Consider the recurring contract* $\mathbf{Rep}(C_3)$*, where the tenant must pay rent (or a fine) every month.*

$$\mathbf{Rep}(C_3) = \mathbf{Rep}(\mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})).$$

Trace Analysis. *Consider a trace in which the tenant pays in Month 1 but fails to pay in Month 2, instead occupying the property.*

- ***Step 1 (*A_1*):*** *The tenant pays. The instance of* C_3 *for Month 1 is discharged. Due to the repetition operator, the residual is* $\varepsilon_C; \mathbf{Rep}(C_3) \equiv \mathbf{Rep}(C_3)$*. The contract effectively "resets" for the next month.*
- ***Step 2 (*A_2*):*** *The tenant occupies but does not pay. The instance of* C_3 *for Month 2 violates the condition. Unlike Step 1, the residual does not reset cleanly. Instead, the violated obligation transforms into its reparation* $\mathbf{O}_1(\mathsf{PAY\_F})$*, which must be fulfilled in the* next *step (Month 3), alongside the continuing repetition* $\mathbf{Rep}(C_3)$*.*

This results in an accumulation of duties: the fine from Month 2 and the new rent for Month 3.

While repetitions capture simple recurring duties, more complex contracts are often bound by conditions. The following analysis studies how the *CPM* andles *guarded contracts*, in which the outer structure (the guard) and the inner structure (the obligations) evolve independently until a termination event occurs.

Example 67 (Progression of Guarded Contracts)**.** *A guarded contract that persists until a termination notice (*$\mathsf{Notif\_T}$*) is issued is examined.*

$$\lceil \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)} \rceil \mathbf{Rep}(C_3)$$

Trace 1: Successful Termination. *The tenant pays in Month 1 (*A_1*) and issues a termination notice in Month 2 (*A_2*).*

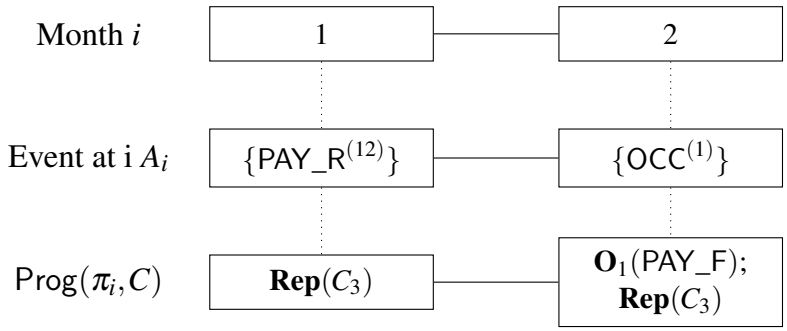


Figure 4.18: Progression on $\mathbf{Rep}(C_3)$ where the obligation is met in the first month but violated in the second.

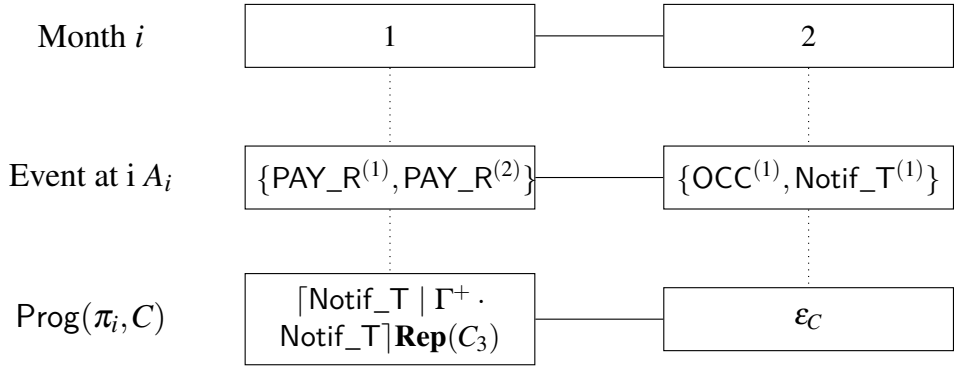


Figure 4.19: Progression on guarded contract where the termination notice at step 2 discharges the contract.

- *At $i = 1$, the event A_1 satisfies the inner contract C_3 (rent paid), but does not satisfy the guard (no notice). The residual is the guarded repetition.*
- *At $i = 2$, the event A_2 contains* Notif_T. *This satisfies the guard expression. The CPM mmediately reduces the entire contract to ε_C, signifying the contract has ended.*

Trace 2: Pending Guard with Internal Violation. *In this scenario, the tenant pays in Month 1 but fails to pay in Month 2 (and gives no notice).*

- *At $i = 2$, the guard is* not *satisfied.*
- *Simultaneously, the inner contract $\mathbf{Rep}(C_3)$ processes the event. Since rent was not paid, the inner contract evolves into a reparation state ($\mathbf{O}_1(\mathsf{PAY\_F})$).*
- *The resulting residual is a guarded reparation: $\lceil \ldots \rceil(\mathbf{O}_1(\mathsf{PAY\_F}); \mathbf{Rep}(C_3))$.*

This illustrates how the CPM aintains the "wrapper" (the guard) while the content inside (the obligations) evolves and accumulates violations independently.

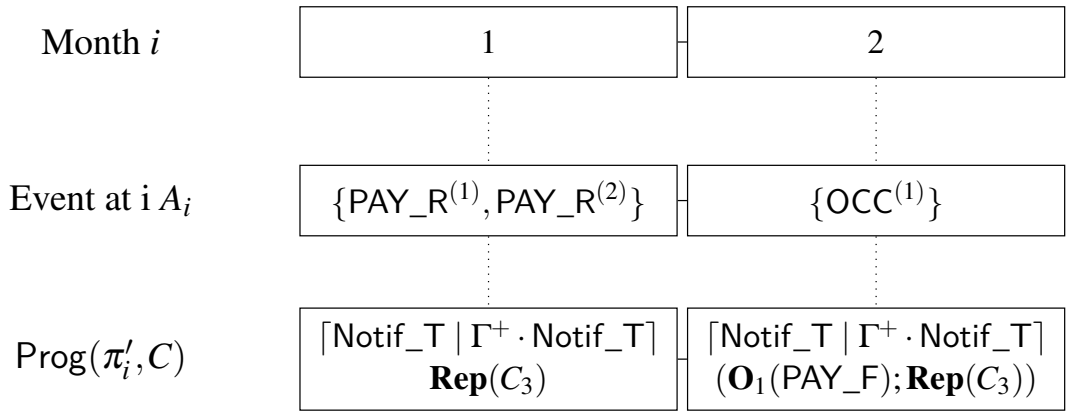


Figure 4.20: Progression on a guarded contract where the guard is not satisfied, and the inner contract triggers a reparation.

We now turn to the notion of reachable residual contracts. This construction is essential for monitor synthesis under the quantitative semantics introduced above. While the progression function describes how a single event advances a contract, monitor construction requires a global view of all contract states that may arise under arbitrary traces. The reachable residual contract set captures precisely this state space.

Definition 65 (Reachable Residual Contract Set from a Contract)**.** *Let C be a contract from* TACNL*, The* reachable residual contract set *of C, written* $\mathsf{RRC}(C)$*, is the set from* $2^{\mathsf{TACNL} \cup \varepsilon_C}$ *capturing all the possible residual contracts obtainable from a contract C by any trace $\pi \in \Gamma^*$ using the progress function*

$$\mathsf{RRC}(C) := \{C_r \mid \exists \tau \in \Gamma^* : \mathsf{Prog}(\pi, C) = C_r\}.$$

Claim 1 (Finiteness of Reachable Residual Contracts)**.** *For every contract $C \in$ TACNL, the reachable residual contract set* $\mathsf{RRC}(C)$ *is finite.*

Justification (informal). The claim follows from the finiteness of the syntactic structure of C and the fact that the progression function Prog rewrites contracts using only subcontracts of C, residuals of regular expressions appearing in C, and the distinguished terminal contract ε_C. No new contract structure is generated beyond these components.

We plan to make this argument constructive by defining, as future work, a function that computes $\mathsf{RRC}(C)$ exhaustively and terminates after finitely many steps. This function will form the basis for an explicit monitor construction and for complexity bounds on the quantitative semantics.

4.7.2 Quantitative Monitoring of Contract Compliance

To define a quantitative violation semantics that persists beyond the first violation prefix, we rely on the *Contract Progress function* as the underlying state-transition mechanism. The progression function Prog dynamically evolves the contract after each observation, producing a sequence of *residual contracts* that represent the exact normative state at each time step. By updating the contract state incrementally, we ensure that the violation score for any given event is calculated strictly against the specific literals in force at that moment, accounting for all prior satisfactions, discharges, or triggered reparations. This updated residual is then propagated to the subsequent evaluation step.

As an optimization, we define a helper function to evaluate only the currently enforced literal for a given event, rather than re-traversing the whole contract structure.

Definition 66 (Literal Head)**.** *Let ε_ℓ be a distinguished symbol denoting the absence of an immediately available literal. The* literal head *function*

$$\mathsf{LH} : \mathsf{TACNL} \to (\ell \cup \{\varepsilon_\ell\}),$$

is defined inductively on the structure of contracts as follows:

$$\begin{aligned}
\mathsf{LH}(\ell) &:= \ell \\
\mathsf{LH}(C;C') &:= \mathsf{LH}(C) \\
\mathsf{LH}(C \blacktriangleright C') &:= \mathsf{LH}(C) \\
\mathsf{LH}(\langle\langle re \rangle\rangle C) &:= \begin{cases} \varepsilon_\ell & \text{if } re \neq \varepsilon, \\ \mathsf{LH}(C) & \text{if } re = \varepsilon. \end{cases} \\
\mathsf{LH}(\lceil re \rceil C) &:= \begin{cases} \mathsf{LH}(C) & \text{if } re \neq \varepsilon, \\ \varepsilon_\ell & \text{if } re = \varepsilon. \end{cases} \\
\mathsf{LH}(C^n) &:= \mathsf{LH}(C) \\
\mathsf{LH}(\mathbf{Rep}(C)) &:= \mathsf{LH}(C).
\end{aligned}$$

We now define the quantitative violation semantics based on two components: 1. the progress monitor, which identifies the remaining parts of a contract to be evaluated, and 2. a scoring mechanism that evaluates each event against the head literal of the residual contract returned by Prog. The final score is the sum of violations for each event along the trace.

Definition 67 (Quantitative Violation Semantics)**.** *Let $\langle A \rangle$ be a single event trace over Γ, π be a (possibly empty) finite trace in Γ^*, and C be a contract from* TACNL*. We define*

the quantitative violation semantics*, denoted by* $[\![]\!]_{qviol} : \Gamma^* \times \mathsf{TACNL} \to \mathbb{N}$*, which maps a trace and a contract to a natural number representing the violation score. The function is defined recursively by evaluating the head of the trace (*$\langle A \rangle$*) and propagating the residual contract to the tail (*π*):*

$$[\![\langle A \rangle \circ \pi, C]\!]_{qviol} := \begin{cases} [\![\langle A \rangle, C]\!]_{qviol} & \text{if } \pi = \langle - \rangle \text{ or} \\ & \mathsf{Prog}(\langle A \rangle, C) = \varepsilon_C, \\ [\![\langle A \rangle, C]\!]_{qviol} + [\![\pi, \mathsf{Prog}(\langle A \rangle, C)]\!]_{qviol} & \text{otherwise.} \end{cases}$$

Here, the instantaneous violation score *for a single event* $\langle A \rangle$ *against a contract* C *is defined inductively on the structure of the contract:*

$$[\![\langle A \rangle, C]\!]_{qviol} := \begin{cases} [\![\langle A \rangle, C_1]\!]_{qviol} + [\![\langle A \rangle, C_2]\!]_{qviol} & \text{if } C = C_1 \wedge C_2, \\ 0 & \text{if } \neg(\langle A \rangle \models_{\perp^t} \mathsf{LH}(C)) \\ & \text{or } \mathsf{LH}(C) = \varepsilon_\ell, \\ 1 & \text{if } \langle A \rangle \models_{\perp^t} \mathsf{LH}(C). \end{cases}$$

Intuitively, the formula $[\![\langle A \rangle \circ \pi, C]\!]_{\text{qviol}}$ treats the contract execution as a path-accumulation problem. At every time step, the function performs three operations:

1. **Snapshots the penalty:** It calculates $[\![\langle A \rangle, C]\!]_{\text{qviol}}$, asking "Given the current literals from C, does the current event A violate any of them?" This is a stateless check based purely on the structure of C at that instant.
2. **Updates the state:** It computes $\mathsf{Prog}(\langle A \rangle, C)$, effectively moving the contract pointer forward (e.g., from a paid obligation to the next month's rent, or from a violated duty to a reparation).
3. **Accumulates:** It adds the snapshot penalty to the result of the recursive call on the remaining trace using the *new* state.

Explicit handling of Sequence and Reparation is managed by the contract progress function, which ensures that "zero-delay" transitions are penalized correctly. For instance, if a contract requires C_1 then C_2, and an event discharges C_1 but violates C_2 in the same step, the summation logic ($[\![\langle A \rangle, C_1]\!]_{\text{qviol}} + [\![\langle A \rangle, C_2]\!]_{\text{qviol}}$) ensures the violation of C_2 is not ignored simply because it appeared in a continuation.

The definition of the instantaneous score $[\![\langle A \rangle, C]\!]_{\text{qviol}}$ rests on distinguishing between **concurrent** (parallel) obligations and **structural** (atomic) constraints. It decouples the measurement of the "volume" of non-compliance from the binary verification of specific rules.

Additivity of Concurrency. The case $[\![\langle A\rangle, C_1 \wedge C_2]\!]_{\text{qviol}} := [\![\langle A\rangle, C_1]\!]_{\text{qviol}} + [\![\langle A\rangle, C_2]\!]_{\text{qviol}}$ captures the "width" of the violation as shown in Figure Figure 4.14. In normative systems, a conjunction represents distinct, independent obligations that are active simultaneously. By summing the scores, the function ensures that the penalty is proportional to the number of distinct parallel duties neglected in a single instant, preventing "violation masking" where a single boolean verdict could hide multiple breaches.

Binary Structural Verdict. For constructs that are not distinct parallel duties (such as literals, sequences, or reparations), the definition relies on the binary tight violation relation ($\models_{\perp^t}$). This captures the "existence" of a fault in a non-decomposable structure. For example, a single atomic duty ($\mathbf{O}(a)$) can only be violated once per step.

Separation of State and Score. This approach assumes that the complexity of temporal evolution is handled by Prog, while $[\![]\!]_{\text{qviol}}$ handles the instantaneous cost. By reducing non-conjunction cases to a simple check ($\langle A\rangle \models_{\perp^t} C$), the definition asserts that scoring is local (checking if the current active node is broken), while progression is temporal (handling the flow from one obligation to the next).

We summarize these properties in the following theorem, which establishes that the quantitative score is a monotonically increasing function that acts as a "super-set" of the binary violation semantics. While a binary trace might be "Satisfied" (via reparation), the quantitative score reveals the true cost of that path.

Theorem 13 (Consistency of Quantitative and Tight Semantics)**.** *For any contract C and finite trace π:*

1. ***Zero Score Implications:*** *If $[\![\pi, C]\!]_{qviol} = 0$, then exactly one of the following three disjoint cases holds:*
   a) $\pi \models_{\top^t} C$ *if and only if* $\mathsf{Prog}(\pi, C) = \varepsilon_C$ *and* $\forall k < |\pi| - 1, \mathsf{Prog}(\pi_k, C) \neq \varepsilon_C$.
   b) $\pi \models_{\top^p} C$ *if and only if* $\mathsf{Prog}(\pi, C) = \varepsilon_C$ *and* $\exists k < |\pi| - 1$ *such that* $\mathsf{Prog}(\pi_k, C) = \varepsilon_C$.
   c) $\pi \models_{?} C$ *if and only if* $\mathsf{Prog}(\pi, C) \neq \varepsilon_C$.
2. ***Non-Zero Score Implications:*** *If $[\![\pi, C]\!]_{qviol} \neq 0$, then:*
   a) $\pi \models_{\perp^t} C$ *if and only if* $[\![\pi, C]\!]_{qviol} = 1$ *and* $[\![\pi_{|\pi|-1}, C]\!]_{qviol} = 0$.
   b) $\pi \models_{\perp^p} C$ *if and only if* $[\![\pi, C]\!]_{qviol} > 1$ *or (*$[\![\pi, C]\!]_{qviol} = 1$ *and* $[\![\pi_{|\pi|-1}, C]\!]_{qviol} = 1$*).*

3. ***Reparation Cost:*** *If π satisfies C strictly through a reparation mechanism (i.e., $\pi \models_{\top^t} C$ but primary obligations failed), then $[\![\pi, C]\!]_{qviol} > 0$.*

Proof. We prove the implications by structural induction on the trace π and the contract C, utilizing the definitions of the quantitative function $[\![]\!]_{\text{qviol}}$ and the contract progression Prog.

1. Zero Score Implications ($[\![\pi, C]\!]_{\text{qviol}} = 0$). Assume $[\![\pi, C]\!]_{\text{qviol}} = 0$. By the definition of the cumulative score, this implies that for all steps $i < |\pi|$, the instantaneous penalty is zero: $[\![\langle A_i \rangle, \mathsf{Prog}(\pi_i, C)]\!]_{\text{qviol}} = 0$. Consequently, no tight violation has occurred at any step. The contract state evolves purely via Prog without triggering any penalty clauses.

   a) Tight Satisfaction ($\models_{\top^t}$).

   - ($\Rightarrow$) Assume $\mathsf{Prog}(\pi, C) = \varepsilon_C$ and for all strict prefixes π', $\mathsf{Prog}(\pi', C) \neq \varepsilon_C$. The condition $\mathsf{Prog}(\pi, C) = \varepsilon_C$ indicates that the contract has been fully discharged. Since the score is 0, this discharge was not achieved via a violation-triggered path (e.g., a reparation where the primary failed). The absence of ε_C in prior prefixes ensures that this is the *first* moment of discharge. By Definition 48 (Tight Satisfaction), the first prefix to fully satisfy the obligations corresponds to $\models_{\top^t}$.
   - ($\Leftarrow$) If $\pi \models_{\top^t} C$, the progression must reach ε_C exactly at π. Since it is a *tight* satisfaction, no proper prefix could have satisfied it (reached ε_C) earlier.

   b) Post Satisfaction ($\models_{\top^p}$).

   - The condition $\exists k < |\pi|$ such that $\mathsf{Prog}(\pi_k, C) = \varepsilon_C$ implies that the contract was already discharged at a previous step k.
   - By Definition 46, $\pi \models_{\top^p} C$ holds if there exists a strict prefix that tightly satisfies C. Since the score is 0, the path to k was compliant. Thus, the state remains ε_C for the remainder of the trace, maintaining the $\models_{\top^p}$ status.

   c) Pre Satisfaction ($\models_?$).

   - Assume $\mathsf{Prog}(\pi, C) \neq \varepsilon_C$. Since $[\![\pi, C]\!]_{\text{qviol}} = 0$, no violation has occurred. However, the contract has not reduced to the empty contract ε_C, meaning active obligations remain.
   - This satisfies the definition of $\models_?$: the trace is neither satisfied ($\not\models_{\top^t} / \not\models_{\top^p}$) nor violated ($\not\models_{\bot^t} / \not\models_{\bot^p}$). It is effectively "pending."

2. Non-Zero Score Implications ($[\![\pi, C]\!]_{\text{qviol}} > 0$). Assume $[\![\pi, C]\!]_{\text{qviol}} > 0$. This implies $\exists i$ such that $[\![\langle A_i \rangle, C_i]\!]_{\text{qviol}} > 0$.

a) Tight Violation ($\models_{\perp^t}$).

- We consider the case where $[\![\pi, C]\!]_{\text{qviol}} = 1$, the score of the immediate prefix is 0 and let $n = |\pi|$.
- $[\![\pi_{n-1}, C]\!]_{\text{qviol}} = 0$ implies that for all previous steps, the contract was in a compliant state ($\models_?$).
- The jump to $[\![\pi, C]\!]_{\text{qviol}} = 1$ implies that the instantaneous score at the last step $[\![\langle A_{n-1}\rangle, \mathsf{Prog}(\pi_{n-1}, C)]\!]_{\text{qviol}} = 1$.
- A positive instantaneous score corresponds to a tight violation of the active residual contract.
- Since this is the *first* non-zero score, it corresponds to the *first* prefix that triggers a violation. This corresponds to the definition of $\pi \models_{\perp^t} C$.

b) Post Violation ($\models_{\perp^p}$).

- The condition $[\![\pi, C]\!]_{\text{qviol}} > 1$ or ($[\![\pi, C]\!]_{\text{qviol}} = 1$ and $[\![\pi_{n-1}, C]\!]_{\text{qviol}} = 1$) implies that the violation score did not originate purely at the current step (or if it did, it was cumulative).
- Specifically, if $[\![\pi_{n-2}, C]\!]_{\text{qviol}} \geq 1$, then a violation occurred strictly in the past.
- By Definition 46, if a strict prefix tightly violated the contract ($\models_{\perp^t}$), the current trace is in $\models_{\perp^p}$. The non-zero score is carried forward monotonically.

c) Reparation Cost.

- Consider a contract $C_{primary} \blacktriangleright C_{repair}$.
- If π satisfies this strictly through the reparation mechanism, it means π did *not* satisfy $C_{primary}$.
- By the definition of reparation progression, the transition to C_{repair} occurs only if $\pi \models_{\perp^t} C_{primary}$.
- By the definition of the instantaneous scoring function for reparation, $[\![\langle A\rangle, C_{primary} \blacktriangleright C_{repair}]\!]_{\text{qviol}} = 1 + \ldots$ when the primary violates.

Therefore, the path involving the repair accumulates a score of at least 1 (the penalty for breaking the primary), confirming $[\![\pi, C]\!]_{\text{qviol}} > 0$.

□

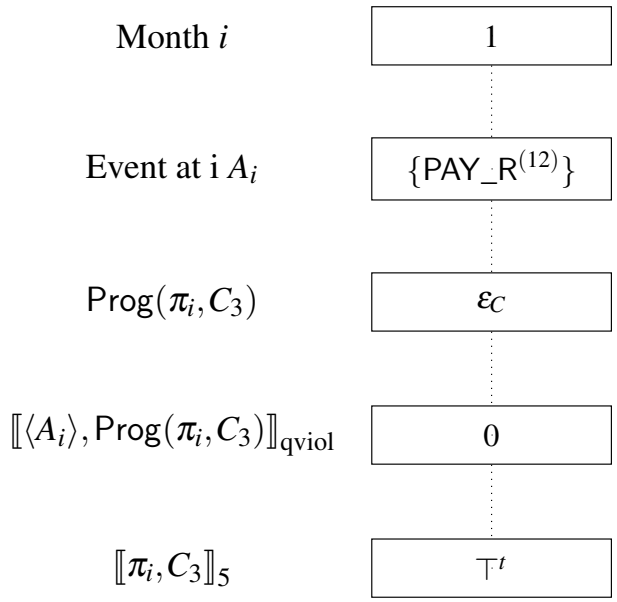


Figure 4.21: Pointwise valuation of the progress (Prog), violation score ($[\![]\!]_{\text{qviol}}$), and tight satisfaction ($[\![]\!]_5$) for the contract $C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$ under the trace $\pi = \langle A_1 \rangle$.

This theorem highlights the utility of the quantitative approach for post-hoc analysis: distinguishing between a "perfect" execution (Score 0) and a "compliant but costly" execution (Score > 0, e.g., paying fines), a distinction lost in the binary $\models_{\top^t}$ verdict.

To illustrate the lemma and the violation semantics, we study several examples:

Example 68 (Reparation: Tight vs. Quantitative Evaluation (Extension of Example 65, Figure 4.16, and Figure 4.17))**.** *We revisit Example 65 and explicitly evaluate both the forward-looking tight semantics and the quantitative violation semantics at each step.*

Let

$$C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F}).$$

Trace $\pi = \langle A_1 \rangle$ with $A_1 = \{\mathsf{PAY\_R}^{(12)}\}$.

- ***Progress:*** $\mathsf{Prog}(\langle A_1 \rangle, C_3) = \varepsilon_C$.
- ***Tight semantics:*** $\langle A_1 \rangle \models_{\top^t} C_3$.
- ***Quantitative score:*** $[\![\langle A_1 \rangle, C_3]\!]_{qviol} = 0$.

This illustrates immediate satisfaction with zero cost. Figure Figure 4.21 summarizes the pointwise valuation for π.

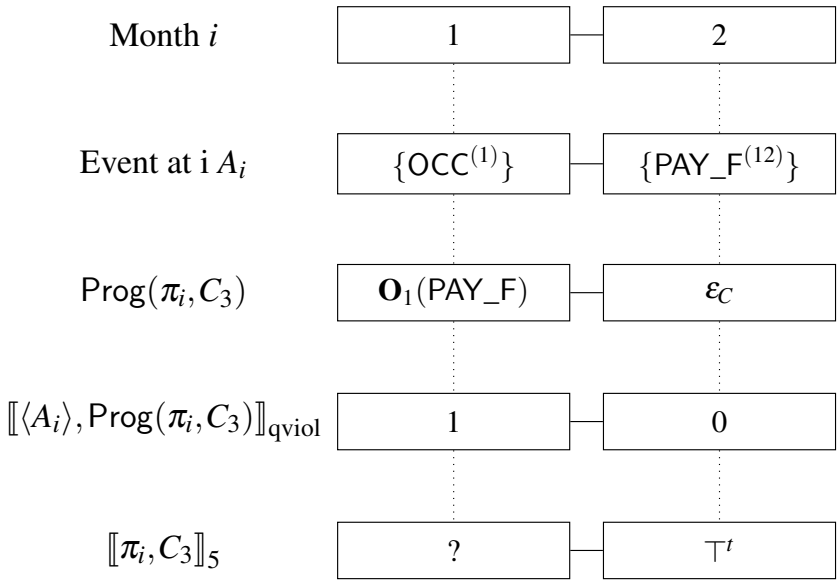


Figure 4.22: Pointwise valuation of the progress (Prog), violation score ($[\![]\!]_{\text{qviol}}$), and tight satisfaction ($[\![]\!]_5$) for the contract $C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$ under the trace $\pi' = \langle A'_1, A'_2\rangle$.

Trace $\pi' = \langle A'_1, A'_2\rangle$ with $A'_1 = \{\mathsf{OCC}^{(1)}\}$ and $A'_2 = \{\mathsf{PAY\_F}^{(12)}\}$.

- ***Step 1:*** $\langle A'_1\rangle \models_{\perp^t} \mathbf{O}_1(\mathsf{PAY\_R})$, *hence* $\mathsf{Prog}(\langle A'_1\rangle, C_3) = \mathbf{O}_1(\mathsf{PAY\_F})$, *and* $[\![\langle A'_1\rangle, C_3]\!]_{qviol} = 1$.
- ***Step 2:*** $\langle A'_2\rangle \models_{\top^t} \mathbf{O}_1(\mathsf{PAY\_F})$, $\mathsf{Prog}(\pi', C_3) = \varepsilon_C$, *and* $[\![\langle A'_2\rangle, \mathbf{O}_1(\mathsf{PAY\_F})]\!]_{qviol} = 0$.

Overall, $\pi' \models_{\top^t} C_3$ *but* $[\![\pi', C_3]\!]_{qviol} = 1$, *illustrating Theorem 13(3). Figure Figure 4.22 summarizes the pointwise valuation for* π'.

Example 69 (Infinite Repetition with Accumulated Cost (Extension of Figure 4.18))**.** *We extend Example (Progression of Infinite Repetition) for*

$$\mathbf{Rep}(C_3) = \mathbf{Rep}(\mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})).$$

Consider the trace $\pi = \langle A_1, A_2\rangle$ *with* $A_1 = \{\mathsf{PAY\_R}^{(12)}\}$ *and* $A_2 = \{\mathsf{OCC}^{(1)}\}$.

- ***Step 1:*** $\langle A_1\rangle \models_{\top^t} C_3$, $\mathsf{Prog}(\langle A_1\rangle, \mathbf{Rep}(C_3)) = \mathbf{Rep}(C_3)$, *and* $[\![\langle A_1\rangle, \mathbf{Rep}(C_3)]\!]_{qviol} = 0$.
- ***Step 2:*** $\langle A_2\rangle \models_{\perp^t} C_3$, $\mathsf{Prog}(\langle A_2\rangle, \mathbf{Rep}(C_3)) = \mathbf{O}_1(\mathsf{PAY\_F}); \mathbf{Rep}(C_3)$, *and* $[\![\langle A_2\rangle, \mathbf{Rep}(C_3)]\!]_{qviol} = 1$.

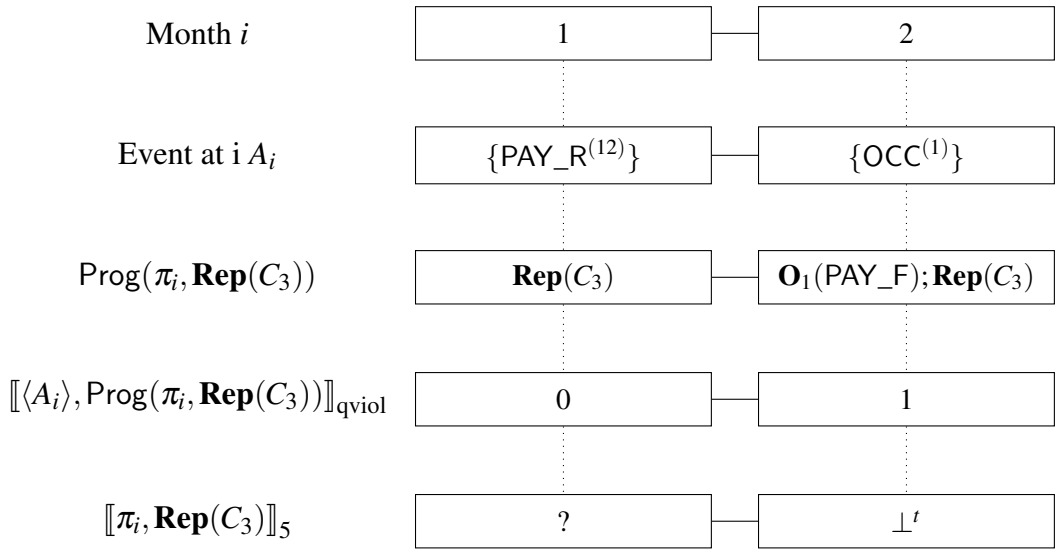


Figure 4.23: Pointwise valuation of the progress (Prog), violation score ($[\![]\!]_{\text{qviol}}$), and tight satisfaction ($[\![]\!]_5$) for the contract $\mathbf{Rep}(C_3)$ under the trace $\pi = \langle A_1, A_2 \rangle$.

Thus, $\pi \models_{\bot^p} \mathbf{Rep}(C_3)$ while $[\![\pi, \mathbf{Rep}(C_3)]\!]_{qviol} = 1$. A longer trace that later satisfies the fine would increase satisfaction without decreasing the accumulated score, illustrating monotonicity. Figure Figure 4.23 summarizes the pointwise valuation for this trace.

Example 70 (Guarded Contract: Divergence Between Status and Cost (Extension of Example Figure 4.19–Figure 4.20))**.** *We extend Example (Progression of Guarded Contracts) for*

$$C := \lceil \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)} \rceil \mathbf{Rep}(C_3).$$

Trace $\pi = \langle A_1, A_2 \rangle$ with $A_1 = \{\mathsf{PAY\_R}^{(12)}\}$ and $A_2 = \{\mathsf{Notif\_T}^{(1)}\}$.

- ***Step 1:*** $\langle A_1 \rangle \models_? C$, $\mathsf{Prog}(\langle A_1 \rangle, C) = C$, *and* $[\![\langle A_1 \rangle, C]\!]_{qviol} = 0$.
- ***Step 2:*** $\langle A_2 \rangle \models_{\top^t} C$, $\mathsf{Prog}(\pi, C) = \varepsilon_C$, *and* $[\![\langle A_2 \rangle, C]\!]_{qviol} = 0$.

Hence, $\pi \models_{\top^t} C$ and $[\![\pi, C]\!]_{qviol} = 0$. Figure Figure 4.24 summarizes the pointwise valuation for π.

Trace $\pi' = \langle A_1, A_2' \rangle$ with $A_2' = \{\mathsf{OCC}^{(1)}\}$.

- ***Step 2:*** $\langle A_2' \rangle \models_{\bot^t} \mathbf{Rep}(C_3)$, $\mathsf{Prog}(\pi', C) = \lceil \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)} \rceil (\mathbf{O}_1(\mathsf{PAY\_F}); \mathbf{Rep}(C_3))$, *and* $[\![\langle A_2' \rangle, C]\!]_{qviol} = 1$.

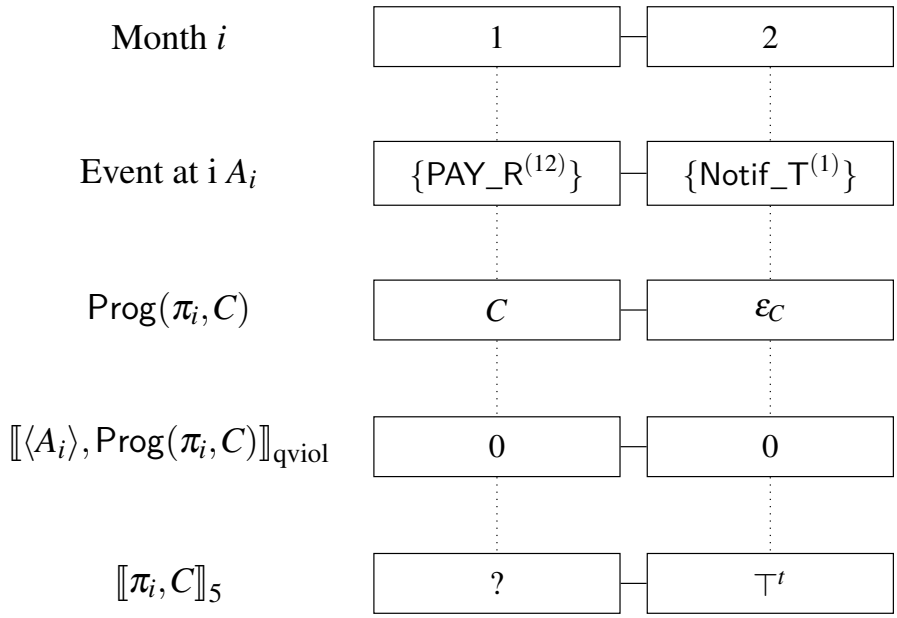


Figure 4.24: Pointwise valuation of the progress (Prog), violation score ($[\![]\!]_{\text{qviol}}$), and tight satisfaction ($[\![]\!]_5$) for the contract $C := \lceil \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)} \rceil \mathbf{Rep}(C_3)$ under the trace $\pi = \langle A_1, A_2 \rangle$.

Thus, $\pi' \models_? C$ in the tight semantics but $[\![\pi', C]\!]_{qviol} = 1$, directly illustrating that quantitative semantics exposes violations masked by guards and reparations. Figure Figure 4.25 summarizes the pointwise valuation for π'.

Conclusion. This section introduced the quantitative violation semantics $[\![]\!]_{\text{qviol}}$ as a trace-level cost measure that complements the tight, three-valued contract semantics. By coupling a local, instantaneous penalty with the residual evolution induced by the progress function Prog, $[\![]\!]_{\text{qviol}}$ cleanly separates temporal state change from scoring, while remaining sensitive to reparations, sequencing, and other structural constructs. Theorem 13 makes this link precise: a zero score characterizes traces that are fully compliant (either tightly satisfied, post-satisfied, or still pending), whereas any positive score pinpoints the presence and persistence of violations, including those that are later repaired and therefore masked at the level of a binary satisfaction verdict. The subsequent examples validate this intuition operationally by tracking Prog, $[\![]\!]_{\text{qviol}}$, and $[\![]\!]_5$ pointwise, showing how the quantitative view supports post hoc audit, comparison of alternative executions, and downstream optimization tasks where "satisfied" is not a sufficient notion of quality.

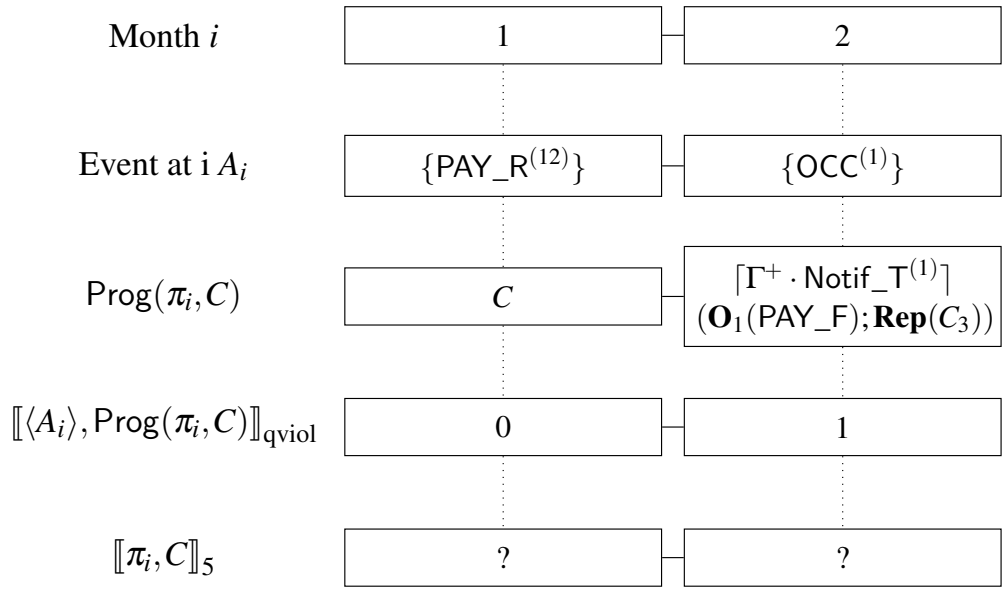


Figure 4.25: Pointwise valuation of the progress (Prog), violation score ($[\![]\!]_{\text{qviol}}$), and tight satisfaction ($[\![]\!]_5$) for the contract $C := \lceil \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)} \rceil \mathbf{Rep}(C_3)$ under the trace $\pi' = \langle A_1, A_2' \rangle$.

4.7.3 Quantitative Violation Monitor Construction

While the forward-looking semantics stops at the first decisive violation, the quantitative semantics requires a monitor that persists on the extended run after the violation, whilst continuing to accumulate violation points over time. To avoid the complexities of using counting machines, we define the *Quantitative Monitor* as a Moore machine whose output alphabet is the set of natural numbers: each state is associated with an instantaneous violation score. Consequently, the overall score for a given trace is derived from the cumulative sum of the machine's outputs at each execution step. Furthermore, this position-based scoring enables precise localization of the time points at which violations occur, thereby enhancing explainability.

Definition 68 (Quantitative Monitor)**.** *The* Quantitative Monitor*, written $\mathcal{M}^{qt}$, is a Mealy machine whose output alphabet is elements from $\mathbb{N}$ representing a score.*

$$\mathcal{M}^{qt} = (Q, q_0, \Gamma, \delta, \lambda_{\mathbb{N}}),$$

where:

1. *Q is the set of states.*
2. *$q_0 \in Q$ is the initial state.*
3. *Γ is the input event alphabet.*

4. $\lambda = Q \times \Gamma \times \to \mathbb{N}$ *is scoring function.*
5. $\delta : Q \times \Gamma \to Q$ *is the transition function.*

Definition 69 (Quantitative Execution Score)**.** *Let* $\mathcal{M}^{qt} = (Q, q_0, \Gamma, \delta, \lambda)$ *be a Quantitative Monitor and let* $\pi = \langle A_0, A_1, \ldots, A_{n-1} \rangle \in \Gamma^*$ *be a finite trace. The execution of* $\mathcal{M}^{qt}$ *on* π *produces an execution* $\langle q_0, A_0, q_1, A_1 \ldots, A_{n-1}, q_n \rangle$ *such that* $q_{i+1} = \delta(q_i, A_i)$ *for all* $0 \leq i < n$*.*

The quantitative score *of the trace* π *on* $\mathcal{M}^{qt}$*, denoted as* $\mathsf{Score}(\mathcal{M}^{qt}, \pi)$*, is defined as the sum of the instantaneous scores of the states visited during the run, excluding the initial state:*

$$\mathsf{Score}(\mathcal{M}^{qt}, \pi) = \sum_{i=0}^{|\pi|-1} \lambda(q_i, \pi(i)).$$

To make the monitor construction constructive and finite, we define it inductively. However, since the states represent residual contracts, the primary challenge lies in bounding the state space. To address this, we leverage the *Reachable Residual Contract Set* ($\mathsf{RRC}(C)$), which we claim to be finite relative to the contract literals and regular expressions.

Definition 70 (Quantitative Monitor Construction)**.** *The* Quantitative Monitor Construction *function, denoted* $\mathsf{QMC}(C)$*, takes a contract* $C \in \mathsf{TACNL}$ *and returns a Quantitative Monitor* $\mathcal{M}^{qt} = (Q, q_0, \Gamma, \delta, \lambda)$ *defined as follows:*

1. $Q := \mathsf{RRC}(C)$.
2. $q_0 := C$.
3. *For any* $q_i \in Q$ *and* $A \in \Gamma$, $\delta(q_i, A) := \mathsf{Prog}(\langle A \rangle, q_i)$.
4. *For any* $q_i \in Q$ *and* $A \in \Gamma$, $\lambda(q_i, A) := [\![\langle A \rangle, q_i]\!]_{qviol}$.

Example 71 (Quantitative Monitor for a Guarded Repetition)**.** *We construct (a finite fragment of) the quantitative monitor* $\mathsf{QMC}(C)$ *for the guarded contract*

$$C := \lceil \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)} \rceil \mathbf{Rep}(C_3), \qquad \textit{where } C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F}).$$

Let $re := \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)}$*. We write* $\lceil re \rceil \mathbf{Rep}(C_3)$ *for the guard instance used below. Starting from the contract as the initial state:*

$$q_0 := \lceil \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)} \rceil \mathbf{Rep}(C_3),$$

The states of the Mealy machine are obtained by exhaustively analyzing which classes of events $A \in \Gamma$ produce a change in the residual via the progress function $\mathsf{Prog}(\langle A \rangle, q)$*, and by recording the corresponding instantaneous cost* $[\![\langle A \rangle, q]\!]_{qviol}$*.*

From q_0, only the satisfaction or violation of the head obligation of C_3 can affect progression, since the guard does not discharge until a termination notification occurs. The head of the repetition body is the primary obligation $\mathbf{O}_1(\mathsf{PAY\_R})$*. Therefore, two event classes are relevant.*

If $A \in \mathsf{PAY\_R}^{(12)}$*, the primary obligation is satisfied. No violation is incurred, hence*

$$[\![\langle A \rangle, q_0]\!]_{qviol} = 0,$$

and progression consumes the obligation and re-enters the repetition:

$$\mathsf{Prog}(\langle A \rangle, q_0) = \mathbf{Rep}(C_3).$$

This residual is represented as state q_2.

If $A \in \overline{\mathsf{PAY\_R}^{(12)}}$*, the primary obligation is violated. This yields a unit penalty*

$$[\![\langle A \rangle, q_0]\!]_{qviol} = 1,$$

and progression activates the reparation clause, producing the residual

$$\mathsf{Prog}(\langle A \rangle, q_0) = \mathbf{O}_1(\mathsf{PAY\_F}); \mathbf{Rep}(C_3),$$

which is represented as state q_1.

From q_1, the head obligation is now the fine obligation $\mathbf{O}_1(\mathsf{PAY\_F})$*. Again, only events that affect its satisfaction or the guard condition are relevant. If* $A \in \mathsf{PAY\_F}^{(12)}$ *and no termination notification occurs, the fine is satisfied with zero cost and progression returns to the repetition residual q_2. If* $A \in \overline{\mathsf{PAY\_F}^{(12)}}$ *and no termination occurs, the fine is violated, yielding cost* 1*, but progression still returns to the repetition, as the fine obligation is consumed after one step.*

If a termination notification $A \in \mathsf{Notif\_T}^{(1)}$ *occurs in state q_1, the guard condition is fulfilled. Progression, therefore, yields the empty contract ε_C, independently of whether the fine is satisfied or violated, while the instantaneous cost is determined by the compliance of the active head obligation at that step.*

From q_2, the repetition phase is stable. Successful payment events $A \in \mathsf{PAY\_R}^{(12)}$ *yield zero cost and leave the residual unchanged, resulting in a self-loop. Failed payment events incur a unit penalty and return progress to the fine residual q_1. As before, the occurrence of a termination notification immediately discharges the contract to ε_C, with*

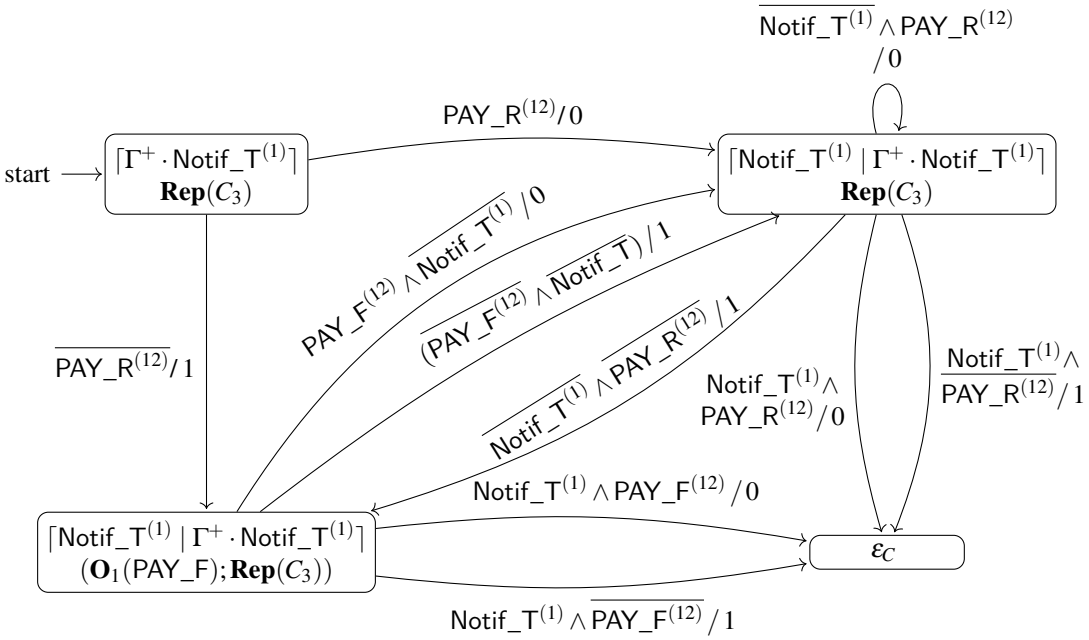


Figure 4.26: The quantitative violation monitor $\mathsf{QMC}(C)$ for $C := \lceil re \rceil \mathbf{Rep}(C_3)$, with $re := \Gamma^+ \cdot \mathsf{Notif\_T}^{(1)}$ and $C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})$. On the transitions:
$\mathsf{PAY\_R}^{(12)} := \{A \in \Gamma \mid \{\mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\} \subseteq A\}$ a shorthand for when the payment successfully occurred.
$\mathsf{Notif\_T}^{(1)} := \{A \in \Gamma \mid \{\mathsf{Notif\_T}^{(1)}\} \subseteq A\}$ for agent (1) sending a termination notification.
$\overline{P} := \{A \in \Gamma \setminus P\}$.
$P \wedge Q := \{A \mid A \in P \cap Q\}$.

the instantaneous cost reflecting whether the payment obligation was satisfied at that final step.

Collecting all distinct residuals reachable from q_0 under these event classes yields exactly the states and transitions shown in Figure Figure 4.26.

We now compute the quantitative execution score for a trace in which the termination notification occurs at the third event. Let

$$\pi = \langle A_1, A_2, A_3 \rangle$$

with $A_1 := \{\mathsf{PAY\_R}^{(1)}, \mathsf{PAY\_R}^{(2)}\}$, $A_2 = \{\mathsf{OCC}^1\}$, *and* $A_3 := \{\mathsf{Notif\_T}^{(1)}, \mathsf{PAY\_F}^{(12)}\}$.

By inspection of Figure Figure 4.26 and by Definition 69, the unique run of $\mathsf{QMC}(C)$ *on* π *is*

$$q_0 \xrightarrow{A_1/0} q_2 \xrightarrow{A_2/1} q_1 \xrightarrow{A_3/0} \varepsilon_C.$$

Therefore, the cumulative quantitative score is

$$\mathsf{score}(\mathsf{QMC}(C),\pi) = \lambda(q_0,A_1) + \lambda(q_2,A_2) + \lambda(q_1,A_3) = 0+1+0 = 1.$$

This shows that although the trace satisfies the guarded contract in the tight semantics due to termination at the third step, the quantitative monitor records the intermediate violation as a persistent cost.

Theorem 14 (Correctness of Quantitative Monitor)**.** *Let C be a contract and $\pi = \langle A_0,\ldots,A_n\rangle$ be a trace. Then the cumulative quantitative violation score is exactly the sum of the monitor outputs:*

$$[\![\pi, C]\!]_{qviol} = \mathsf{Score}(\mathsf{QMC}(C),\pi).$$

Proof. Let $\pi = \langle A_0,\ldots,A_{n-1}\rangle$ be a finite trace of length n. Write $q_0 := C$ and, for each $0 \le i < n$, define the unique run of $\mathsf{QMC}(C)$ on π by

$$q_{i+1} := \delta(q_i,A_i) = \mathsf{Prog}(\langle A_i\rangle, q_i).$$

By definition of the monitor output function, for every i we have

$$\lambda(q_i,A_i) = [\![\langle A_i\rangle, q_i]\!]_{\mathrm{qviol}}.$$

Hence, by Definition 69,

$$\mathsf{Score}(\mathsf{QMC}(C),\pi) = \sum_{i=0}^{n-1} \lambda(q_i,A_i) = \sum_{i=0}^{n-1} [\![\langle A_i\rangle, q_i]\!]_{\mathrm{qviol}}.$$

It remains to show that $[\![\pi, C]\!]_{\mathrm{qviol}}$ expands to the same sum. We prove by induction on n that

$$[\![\pi, C]\!]_{\mathrm{qviol}} = \sum_{i=0}^{n-1} [\![\langle A_i\rangle, q_i]\!]_{\mathrm{qviol}}.$$

Base case ($n = 1$). If $\pi = \langle A_0\rangle$, then by Definition (Quantitative Violation Semantics) $[\![\pi, C]\!]_{\mathrm{qviol}} = [\![\langle A_0\rangle, C]\!]_{\mathrm{qviol}} = [\![\langle A_0\rangle, q_0]\!]_{\mathrm{qviol}}$. This is exactly the required sum.

Inductive step. Assume the claim holds for all traces of length n. Let $\pi' = \langle A_0, \ldots, A_n \rangle$ be a trace of length $n+1$. Unfolding the recursive definition of $[\![]\!]_{\text{qviol}}$ yields

$$\begin{aligned}[\![\pi', C]\!]_{\text{qviol}} &= [\![\langle A_0 \rangle, C]\!]_{\text{qviol}} + [\![\langle A_1, \ldots, A_n \rangle, \mathsf{Prog}(\langle A_0 \rangle, C)]\!]_{\text{qviol}} \\ &= [\![\langle A_0 \rangle, q_0]\!]_{\text{qviol}} + [\![\langle A_1, \ldots, A_n \rangle, q_1]\!]_{\text{qviol}}.\end{aligned}$$

Applying the induction hypothesis to the suffix trace $\langle A_1, \ldots, A_n \rangle$ with initial contract q_1 gives

$$[\![\langle A_1, \ldots, A_n \rangle, q_1]\!]_{\text{qviol}} = \sum_{i=1}^{n} [\![\langle A_i \rangle, q_i]\!]_{\text{qviol}}.$$

Therefore,

$$[\![\pi', C]\!]_{\text{qviol}} = [\![\langle A_0 \rangle, q_0]\!]_{\text{qviol}} + \sum_{i=1}^{n} [\![\langle A_i \rangle, q_i]\!]_{\text{qviol}} = \sum_{i=0}^{n} [\![\langle A_i \rangle, q_i]\!]_{\text{qviol}}.$$

This establishes the inductive claim.

Combining the two displayed equalities, we obtain
$[\![\pi, C]\!]_{\text{qviol}} = \sum_{i=0}^{n-1} [\![\langle A_i \rangle, q_i]\!]_{\text{qviol}} = \mathsf{Score}(\mathsf{QMC}(C), \pi)$. □

Conclusion of the quantitative monitor construction. The monitor construction $\mathsf{QMC}(C)$ gives a finite, executable representation of the quantitative violation semantics whenever the set of reachable residual contracts under Prog is finite. Its key benefit is operational: instead of re-evaluating a contract from scratch on each prefix, we track the evolving residual contract as the current state, and we emit a local score via $\lambda(q, A) = [\![\langle A \rangle, q]\!]_{\text{qviol}}$. The correctness theorem above guarantees that summing these transition-local outputs exactly recovers the trace-level cost $[\![\pi, C]\!]_{\text{qviol}}$. In the next section, we refine this cost into an *attribution* measure by splitting the accumulated score across agents.

4.7.4 Quantitative Blame Semantics

While $[\![\pi, C]\!]_{\text{qviol}}$ reports *how much* a trace deviates from a contract, it does not explain *who* caused the deviation in a multi-agent setting. Quantitative blame addresses this gap by refining each unit violation into an agent-indexed contribution. Concretely, we replace the scalar output alphabet $\mathbb{N}$ by vectors in $\mathbb{N}^2$, where the first component counts violations attributed to agent 1 and the second counts violations attributed to agent 2. The definition below corresponds to the structure of $[\![]\!]_{\text{qviol}}$ and reuses the same progress-based unfolding of residual contracts.

Definition 71 (Quantitative Blame Semantics)**.** *Let* $\pi \in \Gamma^*$ *be a finite trace and* $C \in$ TACNL *be a contract. The* quantitative blame semantics *is a function*

$$[\![\cdot]\!]_{qblame} : \Gamma^* \times \mathsf{TACNL} \to \mathbb{N}^2$$

that maps a trace and a contract to a blame vector (n_1, n_2)*, where* n_i *counts the number of violations attributed to agent i.*

The function is defined recursively on the trace structure as follows:

$$[\![\langle A\rangle \circ \pi, C]\!]_{qblame} := \begin{cases} [\![\langle A\rangle, C]\!]_{qblame} & \text{if } \pi = \langle -\rangle \vee \mathsf{Prog}(\langle A\rangle, C) = \varepsilon_C, \\ [\![\langle A\rangle, C]\!]_{qblame} + & \text{otherwise.} \\ [\![\pi, \mathsf{Prog}(\langle A\rangle, C)]\!]_{qblame} & \end{cases}$$

The instantaneous quantitative blame *for a single event* $\langle A\rangle$ *against a contract* C *is defined by:*

$$[\![\langle A\rangle, C]\!]_{qblame} := \begin{cases} [\![\langle A\rangle, C_1]\!]_{qblame} + [\![\langle A\rangle, C_2]\!]_{qblame} & \text{if } C = C_1 \wedge C_2, \\ (1,0) & \text{if } \langle A\rangle \models_{\bot_1^t} \mathsf{LH}(C), \\ (0,1) & \text{if } \langle A\rangle \models_{\bot_2^t} \mathsf{LH}(C), \\ (0,0) & \text{otherwise.} \end{cases}$$

Additivity and accumulation. The recursive clause uses the usual component-wise addition on $\mathbb{N}^2$: if an event contributes (i, j) at the current step and the suffix contributes (k, l), then their sum is $(i+k, j+l)$. Hence, the blame vector accumulated over a trace counts, for each agent separately, how many step-local violations were attributed to that agent along the unique progress-induced run. This corresponds to the scalar accumulation in $[\![]\!]_{\text{qviol}}$, but preserves per-agent accountability.

Why there is no $(1,1)$ **case.** We do not include a rule for a joint violation vector $(1,1)$ in the definition of $[\![\langle A\rangle, C]\!]_{\text{qblame}}$, such as a hypothetical condition $\langle A\rangle \models_{\bot_{12}^t} \mathsf{LH}(C)$. This is intentional: for the atomic head literal $\mathsf{LH}(C)$, the blame verdicts used in $\models_{\bot_i^t}$ are mutually exclusive in the underlying tight blame semantics, so a single literal cannot be violated by both agents in the same step. When multiple obligations are active concurrently, the conjunction case $C = C_1 \wedge C_2$ already captures multiple violations in one step by summing the vectors returned for each component, and this can yield a total vector whose two components are both positive, but only via *distinct* literals.

Example 72 (Blame Attribution and Quantitative Scores on the Same Traces)**.** *We reuse the contracts and traces from the quantitative violation semantics section and evaluate them simultaneously under the forward-looking blame semantics and the quantitative violation score.*

Recall the contract

$$C_3 := \mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F}),$$

and the permission

$$C_2 := \mathbf{P}_1(\mathsf{OCC}).$$

Trace $\pi = \langle A_1, A_2 \rangle$ with $A_1 = \{\mathsf{OCC}^{(1)}\}$ and $A_2 = \{\mathsf{PAY\_F}^{(12)}\}$. Step 1. *The active obligation is* $\mathbf{O}_1(\mathsf{PAY\_R})$. *Since* A_1 *contains no* $\mathsf{PAY\_R}^{(1)}$*, the obligation is violated. Under the blame semantics,*

$$\langle A_1 \rangle \vDash_{\bot_1^t} \mathbf{O}_1(\mathsf{PAY\_R}),$$

so agent 1 is blamed. Quantitatively,

$$[\![\langle A_1 \rangle, C_3]\!]_{qviol} = 1.$$

Progression activates the reparation:

$$\mathsf{Prog}(\langle A_1 \rangle, C_3) = \mathbf{O}_1(\mathsf{PAY\_F}).$$

Step 2. *The active obligation is now* $\mathbf{O}_1(\mathsf{PAY\_F})$. *Since* $A_2 = \{\mathsf{PAY\_F}^{(12)}\}$*, the obligation is satisfied. No new blame arises, and no quantitative penalty is incurred:*

$$\langle A_2 \rangle \nvDash_{\bot_S^t} \mathbf{O}_1(\mathsf{PAY\_F}) \quad \textit{for all } S, \qquad [\![\langle A_2 \rangle, \mathbf{O}_1(\mathsf{PAY\_F})]\!]_{qviol} = 0.$$

Cumulative result. *The tight semantics yields* $\pi \models_{\top^t} C_3$*, while blame semantics records a single tight violation by agent 1. The quantitative score is obtained by summation:*

$$[\![\pi, C_3]\!]_{qviol} = [\![\langle A_1 \rangle, C_3]\!]_{qviol} + [\![\langle A_2 \rangle, \mathsf{Prog}(\langle A_1 \rangle, C_3)]\!]_{qviol} = 1 + 0 = 1.$$

Trace $\pi' = \langle A_1 \rangle$ with $A_1 = \{\mathsf{OCC}^{(1)}\}$. *Consider now the conjunction $C_2 \wedge C_3$.*

The permission $\mathbf{P}_1(\mathsf{OCC})$ is violated because the subject attempts OCC *without cooperation:*

$$\langle A_1 \rangle \vDash_{\perp_2^t} \mathbf{P}_1(\mathsf{OCC}).$$

Simultaneously, the obligation $\mathbf{O}_1(\mathsf{PAY\_R})$ is violated, yielding

$$\langle A_1 \rangle \vDash_{\perp_1^t} \mathbf{O}_1(\mathsf{PAY\_R}).$$

Hence, blame semantics assigns joint responsibility:

$$\langle A_1 \rangle \vDash_{\perp_{12}^t} (C_2 \wedge C_3).$$

Quantitatively, both conjuncts contribute independently:

$$[\![\langle A_1 \rangle, C_2 \wedge C_3]\!]_{qviol} = [\![\langle A_1 \rangle, C_2]\!]_{qviol} + [\![\langle A_1 \rangle, C_3]\!]_{qviol} = 1 + 1 = 2.$$

Quantitative blame. *By Definition 71, the instantaneous blame at Step 1 is*

$$[\![\langle A_1 \rangle, C_3]\!]_{qblame} = (1,0),$$

since $\langle A_1 \rangle \vDash_{\perp_1^t} \mathbf{O}_1(\mathsf{PAY\_R})$. At Step 2, no violation occurs, hence

$$[\![\langle A_2 \rangle, \mathbf{O}_1(\mathsf{PAY\_F})]\!]_{qblame} = (0,0).$$

Therefore, the cumulative blame vector is obtained by component-wise addition:

$$[\![\pi, C_3]\!]_{qblame} = (1,0) + (0,0) = (1,0).$$

So far, we have introduced two quantitative views of non-compliance. The first one, $[\![\pi, C]\!]_{\text{qviol}}$, gives a single number that counts the number of violations along the trace, without caring who caused them. The second one, $[\![\pi, C]\!]_{\text{qblame}}$, keeps the same idea but splits the count into two parts, one per agent. Since both definitions use the same progress function and add up step-by-step contributions, it is natural to ask whether the split version still matches the original violation semantics total. In other words, does adding up the two blame counters always give back the same score as $[\![\pi, C]\!]_{\text{qviol}}$? The following lemma answers this by proving that the overall quantitative violation score is exactly the sum of the two blame components, so quantitative blame is just a finer view of the same quantity, not a different measure.

Lemma 45 (The relation of the blame score to the violation score)**.** *Let $\pi \in \Gamma^*$ be a finite trace and $C \in \mathsf{TACNL}$ be a contract.*

$$\textit{If } [\![\pi, C]\!]_{qblame} = (n_1, n_2)$$

is the quantitative blame vector for π and C, then the quantitative violation score decomposes as

$$[\![\pi, C]\!]_{qviol} = n_1 + n_2.$$

Proof. We prove the claim by induction on the length of the trace π.

Base case. Let $\pi = \langle A \rangle$ be a single-event trace. By Definitions 71 and (Quantitative Violation Semantics), both $[\![\langle A \rangle, C]\!]_{\text{qviol}}$ and $[\![\langle A \rangle, C]\!]_{\text{qblame}}$ are defined solely from the instantaneous evaluation of the head literal $\mathsf{LH}(C)$.

If $\langle A \rangle$ violates no active literal, then

$$[\![\langle A \rangle, C]\!]_{\text{qviol}} = 0 \quad \text{and} \quad [\![\langle A \rangle, C]\!]_{\text{qblame}} = (0, 0).$$

If $\langle A \rangle$ violates $\mathsf{LH}(C)$ and the violation is attributed to agent i, then

$$[\![\langle A \rangle, C]\!]_{\text{qviol}} = 1 \quad \text{and} \quad [\![\langle A \rangle, C]\!]_{\text{qblame}} = \begin{cases} (1,0) & \text{if } i = 1, \\ (0,1) & \text{if } i = 2. \end{cases}$$

In all cases, $[\![\langle A \rangle, C]\!]_{\text{qviol}} = n_1 + n_2$.

Inductive step. Let $\pi = \langle A \rangle \circ \pi'$ with $\mathsf{Prog}(\langle A \rangle, C) \neq \varepsilon_C$. By the recursive definitions,

$$[\![\pi, C]\!]_{\text{qviol}} = [\![\langle A \rangle, C]\!]_{\text{qviol}} + \left[\!\left[\pi', \mathsf{Prog}(\langle A \rangle, C)\right]\!\right]_{\text{qviol}},$$

and

$$[\![\pi, C]\!]_{\text{qblame}} = [\![\langle A \rangle, C]\!]_{\text{qblame}} + \left[\!\left[\pi', \mathsf{Prog}(\langle A \rangle, C)\right]\!\right]_{\text{qblame}}.$$

Write

$$[\![\langle A \rangle, C]\!]_{\text{qblame}} = (a_1, a_2) \quad \text{and} \quad \left[\!\left[\pi', \mathsf{Prog}(\langle A \rangle, C)\right]\!\right]_{\text{qblame}} = (b_1, b_2).$$

By the induction hypothesis,

$$\left[\!\left[\pi', \mathsf{Prog}(\langle A \rangle, C)\right]\!\right]_{\text{qviol}} = b_1 + b_2,$$

and by the base case,

$$[\![\langle A \rangle, C]\!]_{\text{qviol}} = a_1 + a_2.$$

Therefore,

$$[\![\pi, C]\!]_{\text{qviol}} = (a_1 + a_2) + (b_1 + b_2) = (a_1 + b_1) + (a_2 + b_2) = n_1 + n_2,$$

as required. □

Lemma 46 (Blame Score vs. Forward Blame Semantics)**.** *Let $\pi \in \Gamma^*$ be a finite trace and $C \in \mathsf{TACNL}$ be a contract. If the forward blame monitor assigns blame to a set of agents $S \subseteq \{1,2\}$ on π, then the corresponding component(s) of the persistent quantitative blame vector are non-zero. Formally,*

$$\text{if } [\![\pi, C]\!]_{11} = \bot_S^t \text{ or } [\![\pi, C]\!]_{11} = \bot_S^p$$

then there exist $x, y \in \mathbb{N}$ such that

$$[\![\pi, C]\!]_{qblame} = (x, y) \quad \text{and} \quad \begin{cases} x \neq 0 & \text{if } 1 \in S, \\ y \neq 0 & \text{if } 2 \in S. \end{cases}$$

4.7.5 Quantitative Blame Monitor Construction

The forward-looking blame monitor $\mathsf{BMC}(C)$ identifies the *first* decisive blame frontier and then remains in a post-violation sink. To account for *all* blame occurrences over the whole lifespan of an interaction, we introduce a quantitative blame monitor whose output is a pair of natural numbers counting blame assigned to each agent.

Monitor interface. We fix two agents $\{1,2\}$ and write $\mathbb{N}^2$ for blame vectors. For $\vec{n} = (n_1, n_2)$ and $\vec{m} = (m_1, m_2)$ we write

$$\vec{n} + \vec{m} := (n_1 + m_1,\ n_2 + m_2).$$

Definition 72 (Quantitative blame monitor)**.** *Let C be a contract in* TACNL. *The* quantitative blame monitor *for C is the Mealy machine*

$$\mathsf{QBM}(C) := (Q, q_0, \Gamma, \delta, \lambda_{\mathbb{N}^2}),$$

where:

1. $Q := \mathsf{RRC}(C)$ *is the set of reachable residual contracts.*

2. $q_0 := \mathsf{Prog}(\mathit{emptytrace}, C) = C$.
3. $\delta(q, A) := \mathsf{Prog}(\langle A \rangle, q)$.
4. $\lambda_{\mathbb{N}^2}(q, A) := [\![\langle A \rangle, q]\!]_{\mathit{qblame}}$, *that is, the instantaneous blame vector contributed by the current letter A when the active residual is q.*

Definition 73 (Quantitative blame execution score)**.** *Let* $\mathsf{QBM}(C) = (Q, q_0, \Gamma, \delta, \lambda_{\mathbb{N}^2})$ *and let* $\pi = \langle A_0, \ldots, A_{n-1} \rangle \in \Gamma^*$. *The* quantitative blame score *of* π *on* $\mathsf{QBM}(C)$ *is*

$$\mathsf{Score}(\mathsf{QBM}(C), \pi) := \sum_{i=0}^{n-1} \lambda_{\mathbb{N}^2}(q_i, A_i),$$

where $q_{i+1} := \delta(q_i, A_i)$ *and the sum is component-wise.*

Conformance to the quantitative blame semantics. The next theorem states that the machine-level accumulation coincides with the denotational quantitative blame semantics.

Theorem 15 (Conformance of QBM to quantitative blame semantics)**.** *For every* TACNL *contract C and every finite trace* $\pi \in \Gamma^*$,

$$\mathsf{Score}(\mathsf{QBM}(C), \pi) \;=\; [\![\pi, C]\!]_{\mathit{qblame}}\,.$$

Proof. We proceed by induction on $n := |\pi|$.

Base case $n = 0$. If $\pi = \langle - \rangle$, then by Definition 73 the sum is the neutral element $(0, 0)$. By the definition of $[\![\cdot, \cdot]\!]_{\mathrm{qblame}}$ on the empty trace, $[\![\langle - \rangle, C]\!]_{\mathrm{qblame}} = (0, 0)$. Hence, the equality holds.

Inductive step. Let $\pi = \langle A \rangle \circ \pi'$ with $|\pi'| = n - 1$. The first output produced by $\mathsf{QBM}(C)$ is $\lambda_{\mathbb{N}^2}(q_0, A) = [\![\langle A \rangle, C]\!]_{\mathrm{qblame}}$ by Definition 72. The next state is $q_1 = \delta(q_0, A) = \mathsf{Prog}(\langle A \rangle, C)$. Therefore, unfolding Definition 73 yields

$$\mathsf{Score}(\mathsf{QBM}(C), \pi) = [\![\langle A \rangle, C]\!]_{\mathrm{qblame}} + \mathsf{Score}(\mathsf{QBM}(q_1), \pi').$$

By the induction hypothesis applied to the residual contract q_1, we have

$$\mathsf{Score}(\mathsf{QBM}(q_1), \pi') = [\![\pi', q_1]\!]_{\mathrm{qblame}}\,.$$

Hence

$$\mathsf{Score}(\mathsf{QBM}(C), \pi) = [\![\langle A \rangle, C]\!]_{\mathrm{qblame}} + [\![\pi', \mathsf{Prog}(\langle A \rangle, C)]\!]_{\mathrm{qblame}}\,.$$

Finally, this is exactly the recursive clause of the quantitative blame semantics: it evaluates the head letter against the current residual, then propagates the resulting residual to the tail. Thus $\mathsf{Score}(\mathsf{QBM}(C), \pi) = [\![\pi, C]\!]_{\mathrm{qblame}}$. □

The quantitative blame setting combines the persistence of the quantitative monitors with the responsibility granularity of the blame verdicts. Instead of outputting a Boolean or multi-valued status, the monitor outputs a *blame increment vector* in $\mathbb{N}^2$ at each step, where the i-th component counts how many violations at that step are blamed on agent i. The overall quantitative blame score is obtained by summing these vectors along the run.

Example 73 (Quantitative Blame Monitor for Double Blame)**.** *We construct the quantitative blame monitor for the contract*

$$C := \mathbf{P}_1(\mathsf{OCC}) \wedge \mathbf{O}_1(\mathsf{PAY\_R}),$$

where $\mathbf{P}_1(\mathsf{OCC})$ *models the tenant's power to occupy. As in Example 63 from the forward tight blame monitor, the first step fully determines whether a violation occurs. However, in the quantitative blame view, the transition is labeled with a vector in* $\mathbb{N}^2$ *that counts* how many *violations occurred at that step and* who *is responsible.*

We use the following event classes (as shorthand predicates over a letter $A \subseteq \Gamma$*):*

- $\mathsf{OCC}^{\checkmark}$*: occupation succeeds (tenant attempts and landlord cooperates),*
- $\mathsf{OCC}^{\times}$*: occupation is blocked (tenant attempts, landlord withholds cooperation),*
- $\mathsf{PAY\_R}^{\checkmark}$*: rent payment succeeds (joint execution present),*
- $\mathsf{PAY\_R}^{fail}$*: tenant does not attempt to pay (*$\mathsf{PAY\_R}^{(1)} \notin A$*),*
- $\mathsf{PAY\_R}^{blk}$*: tenant attempts to pay but the landlord blocks cooperation (*$\mathsf{PAY\_R}^{(1)} \in A \wedge \mathsf{PAY\_R}^{(2)} \notin A$*).*

The crucial point is that the conjunction is additive *at a time point. Hence, when both conjuncts are violated in the same step, we add their blame contributions componentwise. In particular, the joint failure* $\mathsf{OCC}^{\times} \wedge \mathsf{PAY\_R}^{fail}$ *yields* $(1,1)$*, while the joint failure* $\mathsf{OCC}^{\times} \wedge \mathsf{PAY\_R}^{blk}$ *yields* $(0,2)$ *since both violations are blamed on agent* 2*.*

4.7.6 Conclusion and Limitations

This section introduced a quantitative extension of the forward-looking tight contract semantics, addressing a fundamental limitation of binary, prefix-closed monitoring. While well suited for early stopping and safety enforcement, binary tight semantics collapses all non-compliant executions into a single failure point. This collapse causes a loss of information essential for post-hoc analysis, auditing, and responsibility assessment. To recover this information, the quantitative framework developed here explicitly separates temporal progression from violation measurement. This separation allows contracts to be

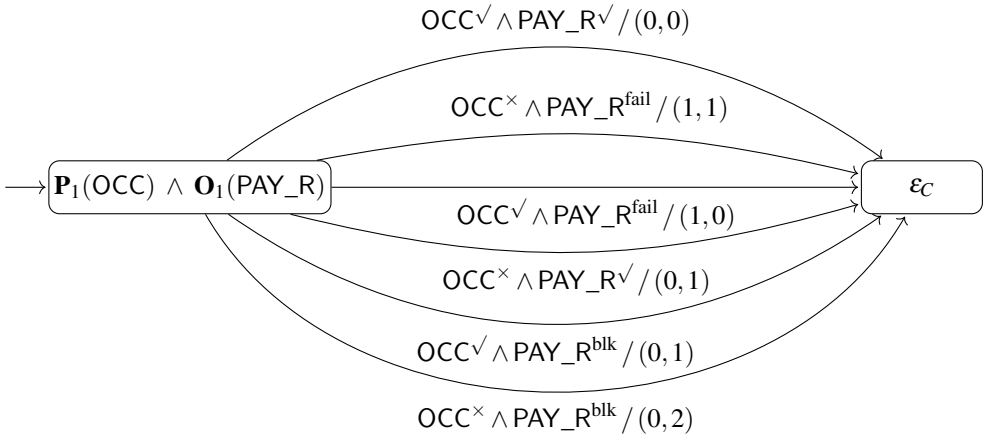


Figure 4.27: Quantitative blame monitor fragment for $\mathbf{P}_1(\mathsf{OCC}) \wedge \mathbf{O}_1(\mathsf{PAY\_R})$. The label $/\,(n_1, n_2)$ is the instantaneous blame increment vector. Note the two joint-failure cases: $(1,1)$ when the violations are blamed on different agents, and $(0,2)$ when both violations are blamed on agent 2 in the same step.

monitored beyond the first violation while preserving a precise account of how and when non-compliance occurs.

The structural backbone of this approach is provided by the Contract Progress Monitor. By evolving contracts into residual contracts after each observed event, the monitor captures the exact normative state at every time step, including sequencing, reparations, repetitions, guards, and triggers. Crucially, this progression is defined independently of scoring. Violations do not halt execution but rather guide structural evolution, such as activating repair clauses or advancing repetitions. This architectural choice ensures that temporal behavior remains faithful to the contract structure while enabling quantitative reasoning on top.

Building directly on this progression, the quantitative violation semantics assigns a local, instantaneous penalty to each event with respect to the currently active residual contract. These penalties are accumulated along the trace to form a quantitative score. Concurrency is treated additively to ensure that parallel obligations cannot mask each other, while structural constructs rely on the underlying tight violation relation. The resulting quantitative score refines classical satisfaction and violation judgments. A zero score characterizes fully compliant executions, including those still pending, while any positive score witnesses the existence and persistence of violations, even when they are later repaired.

A key technical insight is that this framework admits a finite operational realization whenever the set of reachable residual contracts is finite. This reachable set captures exactly the state space induced by progression under arbitrary traces. Finiteness of this set follows from the syntactic finiteness of contracts and the derivative-style evolution of regular ex-

pressions. Such an observation justifies monitor construction and bounds the complexity of quantitative monitoring, remaining independent of trace length.

Overall, the quantitative semantics developed here strictly subsumes the forward-looking tight semantics. While preserving the decisiveness and structural clarity of the original, it augments the logic with a cumulative notion of cost that exposes hidden violations and supports comparison between alternative executions. These capabilities lay the groundwork for agent-level attribution in multi-agent settings. Consequently, the framework is suitable not only for runtime enforcement but also for legal analysis, compliance auditing, and optimization scenarios where binary notions of satisfaction are insufficient.

However, the current scoring mechanism abstracts away various important distinctions found in real-world legal systems. A primary limitation is that the semantics currently conflate the cost of a violation with the cost of a reparation. In our model, a non-zero score simply indicates that the ideal path was not taken, without distinguishing whether the agent is paying a penalty (secondary obligation) or persistently violating the contract. Yet, legal theory emphasizes the fundamental difference between *primary rules* of obligation and *secondary rules* of recognition and adjudication [Har61]. Consistency with this legal convention requires a refinement where the reparation operator $C \blacktriangleright C'$ treats the violation of C differently from the execution of C'. Distinguishing these cases could be achieved by introducing a distinct "reparation score" or by masking the violation score of C when C' is successfully executed, ensuring that "repaired compliance" is semantically distinct from "unrepaired violation."

Furthermore, our framework currently assumes that all failures are directly attributable to the agents. This assumption ignores that real-world contracts regularly experience the *impossibility of performance* due to unenforceable events, such as natural disasters or regulatory changes that render performance illegal [Bai07]. Our model penalizes an agent for failing to act even if the environment prevents the action. Addressing this limitation requires extending the trace model to include a third component: an *environment trace*. Such an extension would allow the semantics to distinguish between unwillingness to perform (fault) and inability to perform (impossibility). This distinction opens the door to new verdict types, such as "shared loss" or "frustration of purpose," where the burden of reparation is distributed between agents rather than assigned to a single defaulter.

Finally, the current reparation operator is agnostic regarding the party responsible for the violation. In a contract $C_1 \blacktriangleright C_2$, the secondary contract C_2 is triggered regardless of which specific clause in C_1 was violated or which agent was responsible. This agnosticism does not reflect legal practice, where the remedy always depends on the specific breach and the party at fault. A necessary extension of this work requires a *parametrized reparation operator* that assigns different secondary obligations depending on the specific agent blamed for the initial violated contract. For instance, if a joint project fails because Agent A did not pay, the reparation should differ from the case where Agent B did not work.

Distinguishing these scenarios requires a richer syntax that propagates blame information into the reparation phase.

4.8 Related Work

Having successfully established the formal semantics and automata-theoretic constructions for TACNL, we now move to discuss the broader research landscape. In this section, we situate our framework against its closest progenitors, specifically STIT logic and the CSL specification language, and contrast it with orthogonal approaches in runtime verification and causal reasoning. This comparison clarifies how TACNL 's synthesis of collaborative agency and state-based reparation addresses a gap in the governance of autonomous digital interactions. We begin by examining the two formalisms that most directly study responsibility in multi-agent settings and contract specifications, before broadening the analysis to complementary notions of causality and algorithmic verification.

4.8.1 Closest Formalisms: STIT and CSL

Inspiration and Context. The development of TACNL has been significantly inspired by the logic of agency STIT [BPX01, Can22] and the contract specification language CSL [HKZ12]. In the following section, we aim to highlight the main differences between these formalisms and TACNL in the most faithful way possible, focusing on the specific divergence in their treatment of agency, reparation, and blame assignment.

TACNL vs. STIT. Both TACNL and STIT model simultaneous interaction through discrete rounds, structurally analogous to a game of "Rock-Paper-Scissors"; however, they diverge significantly in their constraints on agency and reparation. (i) *Agency Structure:* In STIT, interaction is governed by the Independence of Agents condition [BPX01], which treats the choice structure at a moment m as a strict *partition* of histories. This axiom asserts that for any set of agents, the intersection of their respective choices is non-empty, but it restricts each agent to exactly one choice partition per moment. Conversely, TACNL is more flexible regarding action composition: since a round is encoded as a set of moments within a window, agents are permitted to perform multiple actions simultaneously. (ii) *Semantic Faithfulness:* A deeper divergence appears in the handling of reparation. The STIT encoding of reparation via temporal disjunction proves overly permissive, as it lacks the state memory to distinguish between a valid reparation and a spontaneous "fine payment" unrelated to any breach. In contrast, TACNL enforces the strict

procedural semantics of the repair operator ($\blacktriangleright$) through its state-based transition structure (Prog), where the secondary obligation $\mathbf{O}_1(\mathsf{PAY\_F})$ appears as a residual *only* after a transition on the violation branch of $\mathbf{O}_1(\mathsf{PAY\_R})$. This mechanism ensures that reparation actions are not globally admissible propositions, but are causally enabled only by a specific antecedent failure, thereby ruling out anomalous traces like $\langle \mathsf{PAY\_F}, \mathsf{PAY\_F} \ldots \rangle$ where fines are discharged without activation. Consequently, while the TACNL contract $C_7 := \lceil (\Gamma^+ \cdot \mathsf{Notif\_T}) \rceil \mathbf{Rep}((\mathbf{O}_1(\mathsf{PAY\_R}) \blacktriangleright \mathbf{O}_1(\mathsf{PAY\_F})))$ can be approximated in STIT as $\phi_7 := [\{1,2\}\ \mathrm{dstit}\ : ((\mathsf{PAY\_R} \vee \mathsf{PAY\_F})\ \mathcal{U}\ \mathsf{Notif\_T}^{(1)})]$, the latter fails to enforce the causal link between violation and fine.

TACNL vs. CSL. While CSL also formalizes multi-party contracts through an trace-based model focused on run-time monitoring, the two frameworks diverge fundamentally in their action semantics and blame aggregation. Regarding the temporal model, the traces considered in CSL are purely metric timed sequences of events, whereas TACNL employs a periodic synchronized abstraction to group events by period. Although these temporal models are similar modulo the transformations discussed in the preliminaries, the key semantic differences lie in the following points: (i) *Action Semantics:* STIT models the causal "attempt" and potential interference (blocking) through simultaneous choice partitions. In contrast, CSL abstracts away action generation entirely, assuming a fixed input trace of timestamped events. Consequently, CSL does not account for blocked actions; the success of an obligation depends solely on whether the assigned agent performs the action on the trace, ignoring external interference. (ii) *Blame and Quantification:* While STIT evaluates the entire history, allowing for properties like "repeatedly seeing to it", CSL enforces a *fundamental breach* assumption. Although the runtime monitor can assign blame to multiple agents simultaneously (i.e., $|B| > 1$) if violations occur at the exact same timestamp, the semantics are strictly non-quantitative. The monitor stops at the *first* violation instance in the trace, fixing the verdict permanently, and thus cannot capture cumulative penalties or successive violations beyond the first breach.

4.8.2 Broader Landscape and Orthogonal Approaches

Having established the precise relationship between TACNL and its closest formalisms, we now broaden our scope. This section situates the proposed **Two-Agents Collaborative Normative Logic** (TACNL) within the wider landscape of formal methods, deontic logic, and multi-agent systems (MAS) [BGJ16, HKZ12].

The distinguishing thesis is that existing frameworks either idealize away the *bilateral* and *interference-prone* character of contracts, or they fail to translate their rich semantics into lightweight, trace-first monitoring artefacts suitable for runtime accountability [Bro11, Hal16, Sán18].

To make this contrast precise, the discussion is organized along five analytical dimensions:

1. **Temporal scope:** From infinite-trace verification to persistent, prefix-based monitoring.
2. **Normative expressivity:** Handling contrary-to-duty (CTD) patterns and dynamic contract evolution without paradoxes.
3. **Agency and collaboration:** Shifting from zero-sum strategic games to bilateral, synchronized cooperation.
4. **Responsibility attribution:** Contrasting counterfactual causality with trace-based normative accountability.
5. **Monitoring methodology:** Synthesizing automata-theoretic correctness with quantitative stream processing.

Temporal Scope: From Infinite Verification to Persistent Monitoring

Classical verification relies on Linear Temporal Logic (LTL), which interprets specifications over infinite traces and is typically evaluated by model checking [Pnu77]. This is ill-suited to online scenarios where only a finite prefix of a potentially unbounded interaction is available [BLS11, DGV$^+$13].

Runtime verification (RV) adapts temporal reasoning to finite domains, introducing multi-valued semantics for prefixes and monitor synthesis pipelines [Leu11, BLS11]. However, while standard anticipatory semantics typically utilize a three-valued domain $(\top, \bot, ?)$ over finite prefixes, TACNL employs a five-valued semantics. By decomposing the definitive verdicts into temporal variants ($\bot$ is split into permanent violation $\bot^p$ and temporary violation $\bot^t$, and similarly for satisfaction $\top^t, \top^p$), TACNL supports deterministic satisfaction and violation points essential for contract state tracking.

The persistence distinction. Standard RV monitors are often *abortive*: once a property is definitively violated or satisfied, the monitor halts or issues a terminal verdict [Leu11, BLS11]. TACNL instead adopts a *persistent*, contract-centred view. The Contract Progress Function Prog evolves a normative state machine that reacts to violations (e.g., by activating reparations) and continues monitoring the residual contract, thus supporting long-horizon accountability rather than one-shot property checking.

Normative Contrary to Duty handling: From Kripke Models to Trace Decomposition

Standard Deontic Logic (SDL) provides a modal foundation for obligations but notoriously struggles with CTD scenarios, such as the Chisholm paradox, where violations give rise to new obligations that can render the theory inconsistent [Wri51, Chi63, CJ02]. Dyadic deontic logics and Input/Output logic address CTDs by conditioning obligations on explicit contexts or input situations [BFP18, MVDT00].

In parallel, the AI and legal-informatics literature explores more operational perspectives: defeasible deontic reasoning, dynamic deontic operators, and trace-based contract models [PS12, RNA17, HKZ12, Wyn06]. While these approaches bring contracts closer to executable structures, they often emphasize inference over the temporal evolution of a single monitored contract instance. TACNL aligns with the executable perspective but distinctively integrates the *collaborative* and *interference* aspects of the interactions directly into the trace semantics.

Agency: Zero-sum vs Collaborative Interactions

Strategic logics such as ATL and Strategy Logic model coalitional ability and joint moves over game structures and underpin verification tools like MCMAS [AHK02, CHP10, BGJ16, Bel15, ČLMM14]. Their baseline semantics is game-theoretic: a coalition is evaluated by whether it can enforce a goal against the remaining agents or an environment. This perspective fits adversarial settings and many verification scenarios, but it is a poor match for bilateral contracts, where interaction is typically cooperative and compliance is a joint achievement rather than one side "winning".

Even when these logics represent cooperation via coalitions, success is still defined against opposition: if a coalition $C \subseteq \mathcal{C}$ enforces a goal G, then the complement coalition $\mathcal{C} \setminus C$ failed to prevent G. This is not the notion of collaboration required for contracts. In TACNL, we fix exactly two parties, and collaborative actions are interpreted as synchronized attempts: a joint action succeeds only if both parties contribute their required part in the same contractual round. This trace-level view separates non-attempt from blocked attempt and turns interference into observable evidence, which is essential for contractual accountability.

Responsibility: Counterfactual vs. Normative Causality

Structural causal models (SCM) provide a principled account of causality and responsibility based on counterfactual dependence and minimality. In this setting, an agent is

responsible for an outcome if, under a suitable intervention, changing the agent's action would change the outcome [Hal15, Hal16, TSR22].

The causal question addressed by SCM is inherently counterfactual: explaining responsibility by asking what *would* have happened under alternative actions. Such reasoning presupposes an explicit causal model of the environment and its dynamics. In contrast, TACNL adopts a *normative* notion of causality grounded in the rules of the contract itself. Responsibility is not derived from hypothetical alternatives, but from compliance with, or deviation from, the obligations explicitly stated in the contract. A violation is explained by identifying which contractual rule was breached on the observed trace, and blame is assigned to the party whose required contribution was missing, delayed, or blocked according to that rule.

This shift also applies to graded responsibility. While SCM derives degrees of responsibility from counterfactual minimality, TACNL defines degrees of responsibility internally. Grading is computed from the contract structure and its progress semantics—for instance, by counting or weighting violations along the progress-induced run. The resulting notion of responsibility is contract-relative and trace-based: it answers who is responsible, and to what extent, according to the agreed normative commitments rather than an external causal model.

Monitoring Methodology: Correct-by-Construction meets Stream Processing

Principled methodologies for compiling logical specifications into executable monitors have been a central focus of the runtime verification (RV) community. Foundational to this field is the automata-based approach of Leucker and Schallhart, where temporal specifications are systematically translated into monitors with verdicts that are provably aligned with the underlying logic [Leu11]. Generalizing this paradigm to quantitative and data-rich settings is Stream Runtime Verification (SRV). Tools such as Lola, hLola, and TeSSLa support expressive stream transformations and aggregation, making them well suited for monitoring numerical properties over evolving data streams [Sán18, GS18, GS21, CHL+18].

The TACNL framework synthesizes these perspectives through a dual-semantic approach. For its *tight forward semantics*, the language adheres to the classical construction advocated by Leucker and Schallhart: monitors are derived as correct-by-construction Moore machines where every transition is strictly induced by the contract's semantic definition. However, the *quantitative semantics* necessitates a hybrid methodology. While the monitor retains the rigorous structural skeleton of a deterministic state machine, its state valuations integrate stream-based computations similar to SRV. This synthesis allows the monitor to track evolving quantitative data without sacrificing the semantic faithfulness of the underlying automaton.

In conclusion, TACNL occupies a distinct niche within this landscape. While it draws foundational inspiration from the agency axioms of STIT and the trace-based specifications of CSL, it departs from them to address the specific needs of runtime accountability. By prioritizing persistent monitoring over one-shot verification, and normative causality over counterfactual reasoning, TACNL bridges the gap between abstract deontic theory and the operational reality of interference-prone collaborative interactions.

5 Conlusion and Future Works

5.1 Summary of Contributions

This dissertation addresses fundamental challenges in the application of formal methods to the digitalization and automated analysis of normative systems. As laws, regulations, and contracts increasingly govern the behavior of autonomous software agents, the ambiguity of natural language becomes a liability. To mitigate this risk, we demonstrated that rigorous Normative System Engineering can be achieved by restricting logical expressivity to practitioner-oriented fragments. This principled restriction allows for the automation of complex reasoning tasks, specifically: precise conflict detection and rich blame assignment. Without assuming our specific assumption, these reasoning tasks are computationally intractable in more general settings.

The first major contribution, presented in Chapter 3, focused on the static analysis of normative specifications. We introduced the Micro Metric-Time Normative Logic (μMTNL) to formalize the detection of conflicts between time-constrained obligations and prohibitions. By mapping normative requirements to canonical interval sets, we established a precise distinction between deontic conflicts (normative contradictions) and ontic conflicts (physical impossibilities). We further defined a quantitative measure of conflict density and provided an algorithmic procedure for conflict elimination. These tools allow regulators to verify and repair specifications before they are deployed, ensuring that *unmabigous* and easy to plan for.

The second major contribution, detailed in Chapter 4, shifted the focus to the dynamic monitoring of collaborative interactions. We developed the Two-Agents Collaborative Normative Logic (TACNL) to model contracts where compliance depends on the joint actions of multiple parties. A key innovation of this framework is the formal separation of "attempts" from "outcomes," which allows the system to distinguish between genuine non-compliance and failure caused by the other contract party interference. Building on this model, we defined both a forward-looking tight semantics for immediate verdict generation and a quantitative semantics for cumulative blame assessment. The synthesis of these semantics into deterministic monitors provides a verified pipeline for runtime enforcement.

Taken together, these chapters provide a unified methodology for the lifecycle of digital norms. From the static verification of the rule set to the dynamic attribution of blame dur-

ing execution, the formalisms developed here ensure that automated compliance remains transparent, fair, and logically consistent.

5.2 Future Work

The results presented in this dissertation open several avenues for future research, ranging from theoretical extensions to practical applications in formal computational law.

5.2.1 Toward a Unified Formalism (MTTACNL)

The natural progression of this research lies in the synthesis of the two primary formalisms developed in this dissertation. While Chapter 3 established μMTNL for rigorous temporal conflict analysis and Chapter 4 introduced TACNL for blame attribution, their current isolation limits their applicability to scenarios exhibiting both strict timing constraints and cooperative dependencies. To bridge this gap, a primary objective for future work is the development of a unified framework, tentatively styled as the *Metric-Timed Two-Agents Collaborative Normative Logic* (MTTACNL).

This unification requires a fundamental reconstruction of the underlying semantic models. Specifically, the discrete event traces used in our collaborative logic must be replaced by *metric-timed traces* in which every collaborative action and every attempt is associated with a precise real-valued timestamp. On the syntactic level, the logic must support the seamless integration of interval-based constraints with collaborative operators. Such a rich syntax would enable the specification of complex requirements, such as an obligation for "Agent 1 must acomplish action a within 5 time units with the help/without the interference of agent 2."

Operationalizing this unified logic will demand a dual-pronged approach to automated reasoning. First, regarding static analysis, the conflict detection algorithms presented in Chapter 3 must be adapted to account for multi-agent interference. This adaptation would allow regulators to verify not only that time windows are satisfiable but also that no agent is implicitly required to block another to fulfill their own duties. Second, regarding dynamic enforcement, the monitoring infrastructure must be lifted from standard Moore machines to timed automata or similar real-time computational models. Achieving this would provide a comprehensive solution that simultaneously monitors deadline adherence and attributes blame for non-compliance, thereby fully realizing the vision of robust, time-sensitive digital contracts.

5.2.2 Beyond Discrete Actions

Throughout this dissertation, we have operated under the abstraction that actions are atomic, instantaneous events. While discrete actions do occur in practice and many real-world norms can be encoded under this abstraction, numerous normative specifications impose obligations whose satisfaction requires sustained execution over an extended period rather than an instantaneous event. A natural extension of our model is the incorporation of actions with duration, where compliance depends not merely on the occurrence of an event but on the sustained maintenance of a behavior over a prescribed time interval, as suggested by reviewers of the conference version of this work. Pushing this complexity further leads to the domain of *continuous real-time actions*, a challenging counterpart to discrete logic where the system must monitor varying physical signals such as velocity or temperature rather than symbolic transitions. Mastering this continuous domain requires developing elegant abstractions that bridge the gap between normative reasoning and control theory, a synthesis necessary to handle the infinite state space of physical time.

Moving beyond the temporal nature of actions brings us to the fundamental distinction between operational steps and declarative goals. Our current formalism is strictly an *ought-to-do* logic, focusing exclusively on the agents' transitions and interactions. However, many real-world regulations are naturally expressed as *ought-to-be* specifications that constrain the resulting system state rather than the specific method of achievement. For example, a safety regulation may mandate that a room remains below a certain temperature without dictating the specific actions of the cooling system. Capturing the full spectrum of real-life normative specifications therefore requires a hybrid logic capable of reasoning about both the actions performed by agents and the environmental states those actions induce.

List of Figures

List of Tables

Abbreviations

CCS	*c*alculus of *c*ommunicating *s*ystems
CNF	*c*onjunctive *n*ormal *f*orm
CSP	*c*ommunicating *s*equential *p*rocesses
DFA	*d*eterministic *f*inite *a*utomaton
DNF	*d*isjunctive *n*ormal *f*orm
DPNF	*d*isjunctive *p*unctual *n*ormal *f*orm
FCL	*f*ormal *c*ontract *l*anguage
HVAC	*h*eating, *v*entilation, and *a*ir *c*onditioning
LAP	*l*iteral *a*ncestor *p*reservation
LIA	*l*inear *i*nteger *a*rithmetic
LT	*l*ogical-*t*ime projection
LTL	*l*inear *t*emporal *l*ogic
LUCO	*l*owest *u*pper *c*ommon Boolean *o*perator
MUS	*m*inimal *u*nsatisfiable *s*ubset
PC	set of *p*unctual *c*onflicts
PNF	*p*unctual *n*ormal *f*orm
QF-LIA	*q*uantifier-*f*ree *l*inear *i*nteger *a*rithmetic
RL	*r*einforcement *l*earning
SAT	*s*atisfiability
SCCS	*s*ynchronous *c*alculus of *c*ommunicating *s*ystems
SMT	*s*atisfiability *m*odulo *t*heories
ST	*s*ynchronous padding
μMTNL	*m*icro *m*etric-*t*ime *n*ormative *l*ogic

Bibliography

[Aal11] AALST, Wil M. P. d.: *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011

[AD94] ALUR, Rajeev ; DILL, David L.: A Theory of Timed Automata. In: *Theor. Comput. Sci.* 126 (1994), Nr. 2, 183–235. http://dx.doi.org/10.1016/0304-3975(94)90010-8. – DOI 10.1016/0304–3975(94)90010–8

[AHK02] ALUR, Rajeev ; HENZINGER, Thomas A. ; KUPFERMAN, Orna: Alternating-Time Temporal Logic. In: *Journal of the ACM* 49 (2002), Nr. 5

[Ajt94] AJTAI, Miklós: The complexity of the pigeonhole principle. In: *Combinatorica* 14 (1994), Nr. 4, S. 417–433. http://dx.doi.org/10.1007/BF01302964. – DOI 10.1007/BF01302964

[All83] ALLEN, James F.: Maintaining knowledge about temporal intervals. In: *Communications of the ACM* 26 (1983), Nr. 11, S. 832–843

[Bai07] BAIRD, Douglas G.: *The Young Astronomers*. 2007

[BB07] BROERSEN, Jan ; BRUNEL, Julien: 'What I Fail to Do Today, I Have to Do Tomorrow': A Logical Study of the Propagation of Obligations. In: *Computational Logic in Multi-Agent Systems (CLIMA XI) – 11th International Workshop, CLIMA 2007, Proceedings* Bd. 5056, Springer, 2007 (Lecture Notes in Computer Science), S. 82–99

[BBT08] BOELLA, Guido ; BROERSEN, Jan M. ; TORRE, Leendert W. N. d.: Reasoning about Constitutive Norms, Counts-As Conditionals, Institutions, Deadlines and Violations. In: BUI, The D. (Hrsg.) ; HO, Tuong V. (Hrsg.) ; HA, Quang-Thuy (Hrsg.): *Intelligent Agents and Multi-Agent Systems, 11th Pacific Rim International Conference on Multi-Agents, PRIMA 2008, Hanoi, Vietnam, December 15-16, 2008. Proceedings* Bd. 5357. Berlin, Heidelberg : Springer, 2008 (Lecture Notes in Computer Science), 86–97

[BC84] BERRY, Gérard ; COSSERAT, Laurent: The ESTEREL synchronous programming language and its mathematical semantics. In: *International Conference on Concurrency* Springer, 1984, S. 389–448

[BCN+18] BENVENISTE, Albert ; CAILLAUD, Benoît ; NICKOVIC, Dejan ; PASSERONE, Roberto ; RACLET, Jean-Baptiste ; REINKEMEIER, Philipp ; SANGIOVANNI-VINCENTELLI, Alberto L. ; DAMM, Werner ; HENZINGER, Thomas A. ; LARSEN, Kim G.: Contracts for System Design. In: *Found. Trends Electron. Des. Autom.* 12 (2018), Nr. 2-3, S. 124–400

[BDDM04] BROERSEN, Jan ; DIGNUM, Frank ; DIGNUM, Virginia ; MEYER, John-Jules C.: Designing a deontic logic of deadlines. In: *International Workshop on Deontic Logic in Computer Science* Springer, 2004, S. 43–56

[Bel15] BELARDINELLI, Francesco: A logic of knowledge and strategies with imperfect information. 2015. – Forschungsbericht

[BFFR18] BARTOCCI, Ezio ; FALCONE, Yliès ; FRANCALANZA, Adrian ; REGER, Giles: *Lecture Notes in Computer Science*. Bd. 10457: *Introduction to Runtime Verification*. Springer, 2018. http://dx.doi.org/10.1007/978-3-319-75632-5. http://dx.doi.org/10.1007/978-3-319-75632-5

[BFP18] BENZMÜLLER, Christoph ; FARJAMI, Ali ; PARENT, Xavier: A Dyadic Deontic Logic in HOL. In: BROERSEN, Jan M. (Hrsg.) ; CONDORAVDI, Cleo (Hrsg.) ; SHYAM, Nair (Hrsg.) ; PIGOZZI, Gabriella (Hrsg.): *Deontic Logic and Normative Systems - 14th International Conference, DEON 2018, Utrecht, The Netherlands, July 3-6, 2018*, College Publications, 2018, S. 33–49

[BGB+20] BHUIYAN, Hanif ; GOVERNATORI, Guido ; BOND, Andy ; DEMMEL, Sébastien ; ISLAM, Mohammad B. ; RAKOTONIRAINY, Andry: Traffic Rules Encoding Using Defeasible Deontic Logic. In: VILLATA, Serena (Hrsg.) ; HARASTA, Jakub (Hrsg.) ; KREMEN, Petr (Hrsg.): *Legal Knowledge and Information Systems - JURIX 2020: The Thirty-third Annual Conference, Brno, Czech Republic, December 9-11, 2020* Bd. 334, IOS Press, 2020 (Frontiers in Artificial Intelligence and Applications), S. 3–12

[BGJ16] BULLING, Nils ; GORANKO, Valentin ; JAMROGA, Wojciech: Logics for reasoning about strategic abilities in multi-player games. In: *Models of strategic reasoning: logics, games, and communities*. Springer, 2016, S. 93–136

[BK08] BAIER, Christel ; KATOEN, Joost-Pieter: *Principles of model checking*. MIT Press, 2008. – ISBN 978–0–262–02649–9

[BLS11] BAUER, Andreas ; LEUCKER, Martin ; SCHALLHART, Christian: Runtime Verification for LTL and TLTL. In: *ACM Transactions on Software Engineering and Methodology* 20 (2011), Nr. 4, S. 14:1–14:64

[BPX01] BELNAP, Nuel ; PERLOFF, Michael ; XU, Ming: *Facing the Future: Agents and Choices in Our Indeterminist World*. Oxford University Press, 2001

[Bro11] BROERSEN, Jan M.: Modeling Attempt and Action Failure in Probabilistic STIT Logic. In: WALSH, Toby (Hrsg.): *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, IJCAI/AAAI, 2011, S. 792–797

[Brz64] BRZOZOWSKI, Janusz A.: Derivatives of regular expressions. In: *Journal of the ACM (JACM)* 11 (1964), Nr. 4, S. 481–494

[BTV06] BOELLA, Guido ; TORRE, Leendert W. N. d. ; VERHAGEN, Harko: Introduction to normative multiagent systems. In: *Comput. Math. Organ. Theory* 12 (2006), Nr. 2-3, S. 71–79

[Can22] CANAVOTTO, Ilaria: *Where Responsibility Takes You: Logics of Agency, Counterfactuals, and Norms*. Bd. 13228. Springer Nature, 2022

[CC77] COUSOT, Patrick ; COUSOT, Radhia: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: GRAHAM, Robert M. (Hrsg.) ; HARRISON, Michael A. (Hrsg.) ; SETHI, Ravi (Hrsg.): *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, ACM, 1977, S. 238–252

[CES09] CLARKE, Edmund M. ; EMERSON, E. A. ; SIFAKIS, Joseph: Model checking: algorithmic verification and debugging. In: *Commun. ACM* 52 (2009), Nr. 11, S. 74–84

[CGS07] CIMATTI, Alessandro ; GRIGGIO, Alberto ; SEBASTIANI, Roberto: A Simple and Flexible way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In: *Journal on Satisfiability, Boolean Modeling and Computation* 5 (2007), S. 1–27

[Chi63] CHISHOLM, Roderick M.: Contrary-to-Duty Imperatives and Deontic Logic. In: *Analysis* 24 (1963), Nr. 2

[CHL+18] CONVENT, Lukas ; HUNGERECKER, Sebastian ; LEUCKER, Martin ; SCHEFFEL, Torben ; SCHMITZ, Malte ; THOMA, Daniel: *TeSSLa: temporal stream-based specification language*. 2018

[CHP10] CHATTERJEE, Krishnendu ; HENZINGER, Thomas A. ; PITERMAN, Nir: Strategy logic, Elsevier, 2010, S. 677–693

[CJ02] CARMO, José ; JONES, Andrew J.: Deontic logic and contrary-to-duties. (2002), S. 265–343

[ČLMM14] ČERMÁK, Petr ; LOMUSCIO, Alessio ; MOGAVERO, Fabio ; MURANO, Aniello: MCMAS-SLK: A model checker for the verification of strategy logic specifications. In: *International Conference on Computer Aided Verification* Springer, 2014, S. 525–532

[CPS14] CAMILLERI, John J. ; PAGANELLI, Gabriele ; SCHNEIDER, Gerardo: A CNL for contract-oriented diagrams. In: *Controlled Natural Language: 4th International Workshop, CNL 2014, Galway, Ireland, August 20-22, 2014. Proceedings 4*. Berlin, Heidelberg, 2014

[CS63] CHOMSKY, Noam ; SCHÜTZENBERGER, Marcel-Paul: The Algebraic Theory of Context-Free Languages. In: BRAFFORT, P. (Hrsg.) ; HIRSCHBERG, D. (Hrsg.): *Computer Programming and Formal Systems*. Amsterdam : North-Holland, 1963, S. 118–161

[CTGK14] COLOMBO TOSATTO, Silvano ; GOVERNATORI, Guido ; KELSEN, Pierre: Detecting deontic conflicts in dynamic settings. In: *Deontic Logic and Normative Systems: 12th International Conference, DEON 2014, Ghent, Belgium, July 12-15, 2014. Proceedings 12* Springer, 2014, S. 65–80

[DGV+13] DE GIACOMO, Giuseppe ; VARDI, Moshe Y. u. a.: Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In: *Ijcai* Bd. 13, 2013, S. 854–860

[Dij76] DIJKSTRA, Edsger W.: *A Discipline of Programming*. Prentice-Hall, 1976 https://www.worldcat.org/oclc/01958445. – ISBN 013215871X

[Dir63] DIRICHLET, Peter Gustav L.: *Vorlesungen über Zahlentheorie*. Druck und Verlag von Georg Reimer, 1863. – Posthumously edited by Richard Dedekind. First introduced the Schubfachprinzip (pigeonhole principle) in lectures given in 1834

[DK98] DIGNUM, Frank ; KUIPER, Ruurd: Obligations and Dense Time for Specifying Deadlines. In: *Thirty-First Annual Hawaii International Conference on System Sciences, Kohala Coast, Hawaii, USA, January 6-9, 1998*. Los Alamitos, CA, USA : IEEE Computer Society, 1998, 186–195

[DR95] DIEKERT, Volker ; ROZENBERG, Grzegorz: *The book of traces*. World scientific, 1995

[dSS+05] D'ANGELO, Ben ; SANKARANARAYANAN, Sriram ; SÁNCHEZ, César ; ROBINSON, Will ; FINKBEINER, Bernd ; SIPMA, Henny B. ; MEHROTRA, Sandeep ; MANNA, Zohar: LOLA: runtime monitoring of synchronous systems. In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)* IEEE, 2005, S. 166–174

[FFM09] FALCONE, Yliès ; FERNANDEZ, Jean-Claude ; MOUNIER, Laurent: Runtime Verification of Safety-Progress Properties. In: *Runtime Verification (RV 2009)* Bd. 5779, Springer, 2009 (Lecture Notes in Computer Science), S. 40–59

[FH16] FARMER, William M. ; HU, Qian: FCL: A formal language for writing contracts. In: *Quality Software Through Reuse and Integration*. Berlin, Heidelberg : Springer, 2016

[FMGY+24] FENG, Nick ; MARSSO, Lina ; GETIR YAMAN, Sinem ; BAATARTOGTOKH, Yesugen ; AYAD, Reem ; DE MELLO, Victoria O. ; TOWNSEND, Beverley ; STANDEN, Isobel ; STEFANAKOS, Ioannis ; IMRIE, Calum ; RODRIGUES, Genaina N. ; CAVALCANTI, Ana ; CALINESCU, Radu ; CHECHIK, Marsha: Analyzing and Debugging Normative Requirements via Satisfiability Checking. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. New York, NY, USA : Association for Computing Machinery, 2024 (ICSE '24). – ISBN 9798400702174

[FPS09] FENECH, Stephen ; PACE, Gordon J. ; SCHNEIDER, Gerardo: Automatic conflict detection on contracts. In: *International colloquium on theoretical aspects of computing* Springer, 2009

[GHRR07] GOVERNATORI, Guido ; HULSTIJN, Joris ; RIVERET, Régis ; ROTOLO, Antonino: Characterising deadlines in temporal modal defeasible logic. In: *Australasian Joint Conference on Artificial Intelligence* Springer, 2007

[GM06] GOVERNATORI, Guido ; MILOSEVIC, Zoran: A Formal Analysis of a Business Contract Language. In: *Information Systems Frontiers* 8 (2006), Nr. 3, S. 273–302. `http://dx.doi.org/10.1007/s10796-006-9008-8`. – DOI 10.1007/s10796–006–9008–8

[GMS06] GOVERNATORI, Guido ; MILOSEVIC, Zoran ; SADIQ, Shazia: Compliance checking between business processes and business contracts. (2006), S. 221–232

[GR11] GOVERNATORI, Guido ; ROTOLO, Antonino: Justice Delayed Is Justice Denied: Logics for a Temporal Account of Reparations and Legal Compliance. In: LEITE, João (Hrsg.) ; TORRONI, Paolo (Hrsg.) ; ÅGOTNES, Thomas (Hrsg.) ; BOELLA, Guido (Hrsg.) ; TORRE, Leon van d. (Hrsg.): *Computational Logic in Multi-Agent Systems - 12th International Workshop, CLIMA XII, Barcelona, Spain, July 17-18, 2011. Proceedings* Bd. 6814. Berlin, Heidelberg : Springer, 2011 (Lecture Notes in Computer Science), 364–382

[GS18] GOROSTIAGA, Felipe ; SÁNCHEZ, César: Striver: Stream runtime verification for real-time event-streams. In: *International Conference on Runtime Verification* Springer, 2018, S. 282–298

[GS21] GOROSTIAGA, Felipe ; SÁNCHEZ, César: HLola: a very functional tool for extensible stream runtime verification. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* Springer, 2021, S. 349–356

[Hal93] HALBWACHS, Nicolas: *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993

[Hal15] HALPERN, Joseph Y.: Cause, responsibility and blame: a structural-model approach. In: *Law, probability and risk* 14 (2015), Nr. 2, S. 91–118

[Hal16] HALPERN, Joseph Y.: *Actual causality*. MiT Press, 2016

[Har61] HART, Herbert L. A.: *The Concept of Law*. Oxford : Clarendon Press, 1961. – Seminal text introducing the union of primary and secondary rules

[HKT01] HAREL, David ; KOZEN, Dexter ; TIURYN, Jerzy: Dynamic logic. In: *ACM SIGACT News* 32 (2001), Nr. 1, S. 66–69

[HKZ12] HVITVED, Tom ; KLAEDTKE, Felix ; ZALINESCU, Eugen: A trace-based model for multiparty contracts. In: *J. Log. Algebraic Methods Program.* 81 (2012), Nr. 2

[HMU01] HOPCROFT, John E. ; MOTWANI, Rajeev ; ULLMAN, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation*. 2. Boston, MA : Addison-Wesley, 2001. – ISBN 978–0–201–44124–6

[Hoa78] HOARE, Charles Antony R.: Communicating sequential processes. In: *Communications of the ACM* 21 (1978), Nr. 8, S. 666–677

[Hoa93] HOARE, C. A. R.: An Axiomatic Basis for Computer Programming. In: COLBURN, Timothy R. (Hrsg.) ; FETZER, James H. (Hrsg.) ; RANKIN, Terry L. (Hrsg.): *Program Verification - Fundamental Issues in Computer Science* Bd. 14. Springer Netherlands, 1993, S. 83–96

[Hor01] HORTY, John F.: *Agency and Deontic Logic*. Oxford, UK : Oxford University Press, 2001. – 208 S. https://global.oup.com/academic/product/agency-and-deontic-logic-9780195134612. – ISBN 978–0–19–513461–2. – Oxford Philosophical Monographs

[Inc22] INCER, Inigo: *The Algebra of Contracts*, University of California, Berkeley, USA, Diss., 2022. https://www.escholarship.org/uc/item/1ts239xv

[KASL23] KHARRAZ, Karam Y. ; AZZOPARDI, Shaun ; SCHNEIDER, Gerardo ; LEUCKER, Martin: Synchronous Agents, Verification, and Blame - A Deontic View. In: ÁBRAHÁM, Erika (Hrsg.) ; DUBSLAFF, Clemens (Hrsg.) ; TARIFA, Silvia Lizeth T. (Hrsg.): *Theoretical Aspects of Computing - ICTAC 2023 - 20th International Colloquium, Lima, Peru, December 4-8, 2023, Proceedings* Bd. 14446, Springer, 2023 (Lecture Notes in Computer Science), S. 332–350

[KLS21] KHARRAZ, Karam Y. ; LEUCKER, Martin ; SCHNEIDER, Gerardo: Timed dyadic deontic logic. Version: 2021. `http://dx.doi.org/10.3233/FAIA210336`. In: *The 34th International Conference on Legal Knowledge and Information Systems (JURIX'21)* Bd. 346. IOS Press, 2021. – DOI 10.3233/FAIA210336, S. 197–204

[KSL24] KHARRAZ, Karam Y. ; SCHNEIDER, Gerardo ; LEUCKER, Martin: On Conflicts and Satisfiability in Metric Timed Normative Logics. In: SAVELKA, Jaromír (Hrsg.) ; HARASTA, Jakub (Hrsg.) ; NOVOTNÁ, Tereza (Hrsg.) ; MÍSEK, Jakub (Hrsg.): *Legal Knowledge and Information Systems - JURIX 2024: The Thirty-seventh Annual Conference, Brno, Czech Republic, 11-13 December 2024* Bd. 395, IOS Press, 2024 (Frontiers in Artificial Intelligence and Applications), 308–313

[Leu11] LEUCKER, Martin: Teaching runtime verification. In: *International Conference on Runtime Verification* Springer, 2011, S. 34–48

[LS05] LIFFITON, Markus E. ; SAKALLAH, Karem A.: On Finding All Minimally Unsatisfiable Subformulas. In: *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT)* Bd. 3569, Springer, 2005 (LNCS), S. 173–186

[LS08] LIFFITON, Markus E. ; SAKALLAH, Karem A.: Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. In: *Journal of Automated Reasoning* 40 (2008), Nr. 1, S. 1–33

[LS09] LEUCKER, Martin ; SCHALLHART, Christian: A brief account of runtime verification. In: *J. Log. Algebraic Methods Program.* 78 (2009), Nr. 5, S. 293–303

[Mil83] MILNER, Robin: Calculi for synchrony and asynchrony. In: *Theoretical Computer Science* 25 (1983), Nr. 3, S. 267–310

[Mil89] MILNER, Robin: *Communication and Concurrency*. Prentice Hall, 1989

[MS10] MARQUES-SILVA, Joao: Minimal unsatisfiability: Models, algorithms and applications. In: *2010 40th IEEE International Symposium on Multiple-Valued Logic* IEEE, 2010, S. 9–14

[MVDT00] MAKINSON, David ; VAN DER TORRE, Leendert: Input/output logics. In: *Journal of philosophical logic* 29 (2000), Nr. 4, S. 383–408

[Pac20] PACE, Gordon J.: A general theory of contract conflicts with environmental constraints. In: *Legal Knowledge and Information Systems*. IOS Press, 2020, S. 83–92

[Pnu77] PNUELI, Amir: The Temporal Logic of Programs. In: *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977, S. 46–57

[PS12] PRISACARIU, Cristian ; SCHNEIDER, Gerardo: A dynamic deontic logic for complex contracts. In: *J. Log. Algebraic Methods Program.* 81 (2012), Nr. 4, 458–490. http://dx.doi.org/10.1016/j.jlap.2012.03.003. – DOI 10.1016/j.jlap.2012.03.003

[RNA17] ROHANINEZHAD, Mahdi ; NOAH, SA M. ; ARIF, Shereena M.: Defeasible policy language for online social networks, 2017, S. 736

[RS59] RABIN, Michael O. ; SCOTT, Dana: Finite Automata and Their Decision Problems. In: *IBM Journal of Research and Development* 3 (1959), Nr. 2, S. 114–125. http://dx.doi.org/10.1147/rd.32.0114. – DOI 10.1147/rd.32.0114

[Sán18] SÁNCHEZ, César: Online and offline stream runtime verification of synchronous systems. In: *International Conference on Runtime Verification* Springer, 2018, S. 138–163

[Sip12] SIPSER, Michael: *Introduction to the Theory of Computation*. 3. Boston, MA : Cengage Learning, 2012. – ISBN 978–1–133–18779–0

[Tho68] THOMPSON, Ken: Regular Expression Search Algorithm. In: *Communications of the ACM* 11 (1968), Nr. 6, S. 419–422. http://dx.doi.org/10.1145/363347.363387. – DOI 10.1145/363347.363387

[TMRG21] TAMARGO, Luciano H. ; MARTÍNEZ, Diego C. ; ROTOLO, Antonino ; GOVERNATORI, Guido: Time, Defeasible Logic and Belief Revision: Pathways to Legal Dynamics. In: *FLAP* 8 (2021), Nr. 4, 993–1022. https://collegepublications.co.uk/ifcolog/?00046

[TSR22] TRIANTAFYLLOU, Stelios ; SINGLA, Adish ; RADANOVIC, Goran: Actual causality and responsibility attribution in decentralized partially observable markov decision processes. In: *Proceedings of the 2022 AAAI/ACM Conference on AI, Ethics, and Society*, 2022, S. 739–752

[Wri51] WRIGHT, Georg H.: Deontic Logic. In: *Mind* 60 (1951), Nr. 237

[Wyn06] WYNER, Adam Z.: Sequences, obligations, and the contrary-to-duty paradox. In: *International Workshop on Deontic Logic and Artificial Normative Systems* Springer, 2006, S. 255–271

Curriculum Vitae

Name. Karam Younes Kharraz

Date and place of birth. Born in 1989 in Rabat, Morocco

Education.

Master in Foundations of Computer Science and Software Engineering (FIIL), University of Paris-Saclay, Orsay, France	2016–2017
Master of Technology in Networks and Systems, University Hassan I, Faculty of Science and Technology, Settat, Morocco	2011–2013
Bachelor in Mathematics and Computer Science, University Mohamed V, Faculty of Science, Rabat, Morocco	2007–2011
High School Diploma, Lycée Moulay Abdellah, Rabat, Morocco	2005–2007