ETH zürich

ETH Library

Refinement of Legal Specifications in the Lex Programming Language

Master Thesis

Author(s):
Degelo, Jonas

Publication date:
2025

Permanent link:
https://doi.org/https://doi.org/10.3929/ethz-b-000738987

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Refinement of Legal Specifications in the Lex Programming Language

Master's Thesis

Jonas Degelo

May 30th, 2025

Supervisor: Prof. Dr. David Basin

Advisors: François Hublet, Dr. Jakob Merane

Department of Computer Science, ETH Zürich

Abstract

The increasing complexity of digital regulations and the difficulty of ensuring consistent compliance have led to a growing interest in runtime enforcement techniques that automatically verify regulatory compliance. However, these approaches depend on formalized representations of legal norms. Lex is a domain-specific language that provides a highly readable and auditable way to formalize legal specifications. This thesis extends Lex with a notion of refinement, enabling modular specification development through code reuse and incremental adaptation. Refinement allows abstract legal specifications to be concretized for specific use cases while maintaining a clear connection to their original intent.

Contents

Contents iii

1 Introduction 1
1.1 Our Approach . . . 2
1.2 Related Work . . . 2
1.2.1 Legal Formalization . . . 2
1.2.2 Refinement . . . 3
1.2.3 Runtime Enforcement . . . 3
1.3 Contributions . . . 3

2 Preliminaries 5
2.1 Metric First-Order Temporal Logic . . . 5
2.2 Runtime Verification . . . 9
2.2.1 Proactive Runtime Enforcement using MFOTL . . . 10
2.2.2 Enforceable MFOTL . . . 11
2.2.3 Constitutive, Regulative, and Exception Rules . . . 13
2.2.4 Introduction . . . 13
2.2.5 Compilation of CRE[MFOTL] to MFOTL . . . 16
2.3 The Lex Programming Language . . . 16
2.3.1 Lex Syntax . . . 17
2.3.2 Lex Section Labels . . . 19
2.3.3 Lex - E-Mail Marketing Example . . . 19
2.4 Event Systems . . . 22
2.5 Refinement of Event Systems . . . 22
2.6 Legal Background . . . 24

3 Refinement of CRE Enforcement 25
3.1 Introduction . . . 25
3.2 Alternative Enforcement Model . . . 26
3.2.1 System Traces . . . 26

3.2.2 Runtime Enforcement as an Event System 31
3.3 Refinement 35
3.3.1 Enforcement Event System Refinement 35
3.3.2 CRE[MFOTL] Refinement 37

4 Implementation **43**
4.1 Introduction 43
4.2 Refinement of Lex Specifications 43
4.2.1 Event Refinement 44
4.2.2 Rule Strengthening and Weakening 45
4.3 Implementation in the Lex Compiler 46

5 Case Study **47**
5.1 Refinement of the EU ePrivacy Directive in Lex 47
5.1.1 Refinement of a Lex Specification 47
5.1.2 Pretix 48
5.2 Lessons Learned 49

6 Conclusion **51**

Bibliography **53**

Compiled Lex Examples **55**

Additional Proofs **57**
Proof for Lemma 3.26 57
Proof for Lemma 3.27 58

Use of Generative AI Tools **61**

Chapter 1

Introduction

As digital services become increasingly important, a growing number of laws are being introduced to regulate the digital world. One prominent example is the EU's General Data Protection Regulation (GDPR), which aims to protect the privacy of individuals in the European Union. However, empirical evidence indicates that such regulations are frequently disregarded in practice [13].

In order to address this challenge, runtime enforcement has emerged as a promising technique to automatically ensure that a system complies with a given regulation. Runtime enforcement provides automated solutions to ensure Compliance by Design with digital regulations by modifying the system's behavior when required.

The feasibility of such enforcement is contingent upon the existence of automated policies that are sufficiently formal. Nevertheless, legal texts written in natural language do not directly meet this requirement. Lex [12, 8], a programming language developed as a specification language for legal texts, has been shown to maintain a high level of readability while being formal enough to develop policies for runtime enforcement. This intricacy is a primary rationale behind the integration of the refinement concept into Lex, as outlined in this thesis. The term "refinement" is employed to denote both a conceptual framework in which a particular concrete specification is said to "implement" (or be "derived" from) a more abstract specification, as well as the process of deriving such a concrete specification from an abstract one. Refinement allows users to reuse existing Lex specifications and make minimal changes to adapt them to their specific use cases.

1.1 Our Approach

Lex is designed to formalize legal texts for the purpose of using the formalization as a policy for runtime enforcement. Despite its advanced features, formalizing a law in Lex remains a complex task. A foundational component of any Lex specification is the concept of events. These events define a model of the world. In order to ensure the accuracy and simplicity of formalization, it is essential that these events be as general as possible. Conversely, to effectively enforce a given system, the events in the specification must directly correspond to system events. This is a fundamental aspect of the challenge this thesis aims to address. It proposes a comprehensive specification that encompasses both abstract and concrete elements, integrating them to create a cohesive system model. This approach facilitates the mapping of the concrete system model to the abstract specification, thereby enabling a seamless integration of both abstract and concrete components. We apply refinement techniques to enable the reuse of a general, abstract formalization as a baseline policy for different systems. This is achieved by defining a comparatively short refinement specification for each system

1.2 Related Work

1.2.1 Legal Formalization

There exist a multitude of languages and logics for formalizing legal texts. Each with its own strengths and weaknesses.

In recent years, two notable domain-specific languages for legal texts have been introduced. Merigoux et al. introduced Catala [15], a programming language intended for directly transforming existing laws, such as the US tax code, into executable code. Catala is noteworthy for its approach to maintaining the structural hierarchy of laws in its code, as well as its utilization of prioritized default logic [6] as a foundation for addressing exceptions. Van Binsbergen et al. [18] have developed eFlint, another programming language that facilitates the transformation of legal specifications into executable code. eFlint does not inherently maintain a law's hierarchy in its code, because of this it can also be used for other kind of norms. While Catala is not intended to be used for enforcement purposes, eFlint can be used for certain kinds of enforcement. eFlint was for example used to enforce that an actor has permission to perform an action before doing so. But it was not used in a way that would ensure full Compliance by Design as violations were sometimes allowed to happen and then reacted to. Both Catala and eFlint do not directly support refinement of existing specifications, and they further differ from Lex in so far as that Lex specifications are not themselves executable.

1.2.2 Refinement

Refinement of specifications is a common technique in formal methods where one starts with a more general specification and works towards a more concrete specification that implements or fulfills the general specification. Abadi and Lamport [3] have introduced refinement mappings as a technique for proving that one specification implements another specification. Sprenger et al. [17] have used refinement as a method to get a concrete implementation of a program that complies with an abstract specification.

1.2.3 Runtime Enforcement

Schneider [16] has characterized the kind of policies that can be enforced mechanically and he has also given techniques for such mechanical enforcement. Since then a lot of research has been done on the topic of runtime enforcement.

Hublet et al. [9, 10] developed a proactive runtime enforcement algorithm that uses Metric First Order Temporal Logic (MFOTL) [5] as a policy language. Hublet et al. [12] and Degelo [8] have introduced the Lex programming language that compiles to MFOTL and is therefore compatible with these proactive runtime enforcement algorithms.

1.3 Contributions

This thesis presents how support for refinement can be added to the Lex programming language, implementats refinement in Lex and demonstrates the refinement capabilities of Lex on a case study. Lex is a programming language for formalizing legal specifications, developed by Hublet, Degelo et al. [12, 8]. The contributions of this thesis are as followed. In Chapter 3 we introduce a proactive enforcement model based on event systems and prove that it is analogous to the enforcement model of Hublet et al. [9, 10]. We then introduce a set of refinement rules for the CRE logic introduced by Hublet et al. [12] and use our new proactive enforcement model. We show that these refinement rules correspond to refinement as used by Sprenger et al. [17] and Abadi and Lamport [3]. Chapter 4 introduces new language features to Lex that allow CRE refinements to be specified in Lex. We both present the syntax and semantics of these new language features and give an implementation of them. Chapter 5 presents a case study where we apply the refinement features of Lex to an example formalization of the EU ePrivacy Directive [1] and how it can be refined to enforce the ePrivacy Directive on the open source ticketing platform Pretix [2].

Parts of the Lex paper [12] (in particular those pertaining to refinement) have been developed in parallel with this thesis.

Chapter 2

Preliminaries

2.1 Metric First-Order Temporal Logic

Metric First-Order Logic (MFOTL) [5, 7] is a logical system that combines first-order predicate logic with temporal operators such as $\bigcirc$ ("next"), $\bullet$ ("previous"), U ("until"), and S ("since"). MFOTL is frequently utilized as a policy language in runtime verification [9, 12, 5].

Definition 2.1 *An* MFOTL Signature $\Sigma = (\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota)$ *is a quadruple consisting of a value domain* $\mathbb{D}$*, a set of event names* $\mathbb{E}$*, a set of function names* $\mathbb{F}$*, and an arity function* $\iota : \mathbb{E} \cup \mathbb{F} \to \mathbb{N}$.

Definition 2.1 is the the same signature definition as in Hublet et. al [12, Section 2.2.2] and Hublet et. al [9, Section 2].

Example 2.2 *For*

$$\begin{aligned}
\mathbb{D} &= \mathtt{strings} \cup \{\bot, \top\} \cup \mathbb{N}, \\
\mathbb{E} &= \{\mathsf{giveEmailConsent}, \mathsf{revokeEmailConsent}, \mathsf{sednEmail}\}, \\
\mathbb{F} &= \{\mathsf{ageOfUser}\} \\
\iota(\mathsf{giveEmailConsent}) &= 2, \\
\iota(\mathsf{revokeEmailConsent}) &= 2, \\
\iota(\mathsf{sendEmail}) &= 2, \\
\iota(\mathsf{ageOfUser}) &= 1
\end{aligned}$$

$\Sigma = (\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota)$ *is an MFOTL signature.*

Definition 2.3 *An* Event *is a tuple* $(e, (p_1, \ldots, p_k)) \in \mathbb{E} \times \mathbb{D}^k$ *where* $k = \iota(e)$ *and all* $p_i \in \mathbb{D}$.

Example 2.4

$$\mathsf{giveEmailConsent}(\textit{"Alice"}, \textit{"Account Updates"}) \textit{ and}$$
$$\mathsf{revokeEmailConsent}(\textit{"Bob"}, \textit{"Marketing"})$$

are events over the signature Σ from Example 2.2.

Definition 2.5 *A* Database $D \in \mathsf{DB}$ *is a set of events.*

Definition 2.6 *A* Trace $\sigma \in \mathbb{T}$ *is a sequence* $\langle(\tau_i, D_i)\rangle_{1 \leq i \leq k}$, $k \in \mathbb{N} \cup \{\infty\}$ *of timestamps* $\tau_i \in \mathbb{N}$ *and finite databases* $D_i \in \mathsf{DB}$*, where timestamps grow* monotonically *(*$\forall i < |\sigma|.\, \tau_i \leq \tau_{i+1}$*) and* progress *(if* $|\sigma| = \infty$*, then* $\lim_{i \to \infty} \tau_i = \infty$*).*

Definition 2.7 *The set of* Infinite Traces *is denoted by*

$$\mathbb{T}_\omega \subseteq (\mathbb{N} \times \mathsf{DB})^\infty$$

Definition 2.8 *The set of* Finite Traces *is denoted by*

$$\mathbb{T}_f \subseteq (\mathbb{N} \times \mathsf{DB})^*$$

Definition 2.9 *A* time-point $0 \leq i \leq |\sigma|$ *is an index in a trace* σ*.*

Example 2.10

$$\begin{aligned}
\sigma = \langle(0, \quad & \{\mathsf{giveEmailConsent}(\textit{"Alice"}, \textit{"Account Updates"})\}),\\
(1, \quad & \{\mathsf{giveEmailConsent}(\textit{"Bob"}, \textit{"Marketing"})\}),\\
(2, \quad & \{\mathsf{revokeEmailConsent}(\textit{"Alice"}, \textit{"Account Updates"}),\\
& \mathsf{giveEmailConsent}(\textit{"Bob"}, \textit{"Account Updates"})\}),\\
(10, \quad & \emptyset)\rangle
\end{aligned}$$

σ is a finite trace over the signature Σ from Example 2.2 with 4 time-points. The 4 time-points have the timestamps 0, 1, 2, *and* 10.

Traces like Example 2.10 can be seen as representing a system's behavior in a standardized form.

Definition 2.11 *The* Empty Trace *is denoted by* ε*.*

Definition 2.12 *The function* $\mathsf{fts} : \mathbb{T} \to \mathbb{N} \cup \{\bot\}$ *gives the first time stamp in a trace unless it is empty.*

$$\mathsf{fts}(\sigma) := \begin{cases} \bot & \text{if } \sigma = \varepsilon \\ \tau & \text{if } \sigma = (\tau, D) \cdot \sigma' \end{cases}$$

$$\frac{c \in \mathbb{D}}{\Gamma \vdash_{trm} c}\text{Constant} \qquad \frac{v \in \mathsf{Var}}{\Gamma \vdash_{trm} v}\text{Variable}$$

$$\frac{f \in \mathbb{F} \quad t_i \in \mathbb{D} \quad \iota(f) = k}{\Gamma \vdash_{trm} f\,(t_1, \ldots, t_k)}\text{Function}$$

Figure 2.1: MFOTL Term Typing Rules

Definition 2.13 *The function* $\mathsf{lts} : \mathbb{T} \to \mathbb{N} \cup \{\bot\}$ *gives the first time stamp in a trace unless it is empty.*

$$\mathsf{lts}(\sigma) := \begin{cases} \bot & \text{if } \sigma = \varepsilon \\ \tau & \text{if } \sigma = \sigma' \cdot (\tau, D) \end{cases}$$

For the trace σ in Example 2.10 we have $\mathsf{fts}(\sigma) = 0$ and $\mathsf{lts}(\sigma) = 10$.

Definition 2.14 *A* Property *is a subset* $P \subseteq \mathbb{T}_\omega$.

A Property can be induced by some policy φ, we denote this policy by $P(\varphi)$. We say a trace σ is compliant with policy φ if and only if $\sigma \in P(\varphi)$.

Definition 2.15 *The set of* Non-Empty Intervals *over* $\mathbb{N}$ *is denoted by* $\mathbb{I}$.

Figure 2.1 defines the rules for when an MFOTL term is well-typed. And Figure 2.2 defines when an MFOTL formula is well-typed. In particular, these two sets of typing rules ensure that all events and functions have the correct number of arguments. Next we introduce the specific MFOTL grammar used which includes the notable additions of non-recursive let-bindings [19] and aggregations [4]. From here on let Var be a set of variables. The following grammar definition is the same as in Hublet et al. [10, Section 2.2].

Definition 2.16 *Let* $c \in \mathbb{D}$, $x \in \mathsf{Var}$, $f \in \mathbb{F}$, $e \in \mathbb{E}$, *and* $I \in \mathbb{I}$. *Then our* MFOTL Grammar *is defined by the following two rules*

$$\begin{aligned} t &::= \; c \mid x \mid f(t, \ldots, t) \\ \varphi &::= \; \top \mid e(t, \ldots, t) \mid t = t \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists.\, \varphi \mid \bigcirc_I \varphi \mid \bullet_I \varphi \mid \varphi \,\mathsf{U}_I\, \varphi \mid \varphi \,\mathsf{S}_I\, \varphi \\ &\quad\; \mid x \leftarrow \omega(t; x, \ldots, x)\; \varphi \mid \mathsf{let}\, e\,(x, \ldots, x) = \varphi \;\mathsf{in}\; \varphi \end{aligned}$$

Definition 2.17 *A variable* Valuation $v : \mathsf{Var} \to \mathbb{D}$ *assigns a value to a variable.*

Definition 2.18 *The* Evaluation *of well-typed MFOTL terms t under some valuation v is (recursively) defined as followed:*

$$\begin{aligned} \downarrow_v c &:= c && \text{for all constants } c \in \mathbb{D} \\ \downarrow_v x &:= v(x) && \text{for all variables } v \in \mathsf{Var} \\ \downarrow_v f\,(t_1, \ldots, t_k) &:= \hat{f}\,(\downarrow_v t_1, \ldots \downarrow_v t_k) && \text{for all functions } f \in \mathbb{F} \text{ and all terms } t_i \end{aligned}$$

$$\frac{}{\Gamma \vdash \top}\,\text{True} \qquad \frac{\Gamma \nvdash \varphi}{\Gamma \vdash \neg\varphi}\,\text{Not} \qquad \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\,\text{And}$$

$$\frac{e \in \mathbb{E} \quad \Gamma \vdash_{trm} x_i}{\Gamma \vdash e\,(x_1, \ldots, x_k)}\,\text{Event} \qquad \frac{\Gamma \vdash \varphi \quad x \in \mathsf{Var}}{\Gamma \vdash \exists x.\varphi}\,\text{Exists}$$

$$\frac{\Gamma \vdash \varphi \quad I \in \mathbb{I}}{\Gamma \vdash \bigcirc_I \varphi}\,\text{Next} \qquad \frac{\Gamma \vdash \varphi \quad I \in \mathbb{I}}{\Gamma \vdash \bullet_I \varphi}\,\text{Previous}$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi \quad I \in \mathbb{I}}{\Gamma \vdash \varphi \,\mathsf{U}_I\, \psi}\,\text{Until} \qquad \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi \quad I \in \mathbb{I}}{\Gamma \vdash \varphi \,\mathsf{S}_I\, \psi}\,\text{Since}$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma[\mathbb{E} \mapsto \mathbb{E} \cup \{e\}, \iota \mapsto \iota[e \mapsto k]] \vdash \psi \quad \Gamma \vdash_{trm} x_i}{\Gamma \vdash \mathsf{let}\, e\,(x_1, \ldots, x_k) = \varphi \,\mathsf{in}\, \psi}\,\text{Let}$$

Figure 2.2: MFOTL Typing Rules

Definition 2.19 *An MFOTL formula φ is* satisfied *at some time-point i in a trace σ under some valuation v is denoted by $v, i \vDash_\sigma \varphi$.*

If the trace is obvious from context the subscript σ in $\vDash_\sigma$ can be omitted.

Definition 2.20 *The* Semantics of MFOTL *for a fixed, infinite trace σ is defined as followed*

$$\begin{array}{ll}
v, i \vDash \top & \\
v, i \vDash \neg\varphi & \textit{iff } v, i \nvDash \varphi \\
v, i \vDash \varphi \wedge \psi & \textit{iff } v, i \vDash \varphi \textit{ and } v, i \vDash \psi \\
v, i \vDash e(t_1, \ldots, t_k) & \textit{iff } e(\downarrow_v t_1, \ldots, \downarrow_v t_k) \in D_i \\
v, i \vDash \exists x.\varphi & \textit{iff } v[x \mapsto d], i \vDash \varphi \textit{ for some } d \in \mathbb{D} \\
v, i \vDash \bigcirc_I \varphi & \textit{iff } v, i+1 \vDash \varphi \textit{ and } \tau_{i+1} - \tau_i \in I \\
v, i \vDash \bullet_I \varphi & \textit{iff } i > 0 \textit{ and } v, i-1 \vDash \varphi \textit{ and } \tau_i - \tau_{i-1} \in I \\
v, i \vDash \varphi \,\mathsf{U}_I\, \psi & \textit{iff } v, j \vDash \psi \textit{ for some } j \geq i \textit{ with } \tau_j - \tau_i \in I \\
 & \quad \textit{and } v, k \vDash \varphi \textit{ for all } i \leq k < j \\
v, i \vDash \varphi \,\mathsf{S}_I\, \psi & \textit{iff } v, j \vDash \psi \textit{ for some } j \leq i \textit{ with } \tau_i - \tau_j \in I \\
 & \quad \textit{and } v, k \vDash \varphi \textit{ for all } j < k \leq i \\
v, i \vDash \mathsf{let}\, e(x_1, \ldots, x_k) = \varphi \,\mathsf{in}\, \psi & \textit{iff } v, i \vDash_{\sigma\left[\forall j. D_j \mapsto D_j \cup \left\{ e\left(\bar{d}\right) \mid v\left[\bar{x} \mapsto \bar{d}\right], j \vDash \varphi \right\}\right]} \psi
\end{array}$$

Where $c \in \mathbb{D}$, $x \in \mathsf{Var}$, $f \in \mathbb{F}$, $e \in \mathbb{E}$, and $I \in \mathbb{I}$.

We define the following operators based on the basic operators above as "syntactic sugar".

Definition 2.21 MFOTL Syntactic Sugar

$$
\begin{aligned}
\bot &:= \neg\top & \varphi \rightarrow \psi &:= \neg\varphi \vee \psi \\
\varphi \vee \psi &:= \neg(\neg\varphi \wedge \neg\psi) & \varphi \leftrightarrow \psi &:= (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \\
\forall x.\varphi &:= \neg(\exists x.\neg\varphi) & & \\
\Diamond_I\, \varphi &:= \top\ \mathsf{U}_I\ \varphi \quad \textit{(eventually)} & \blacklozenge_I\, \varphi &:= \top\ \mathsf{S}_I\ \varphi \quad \textit{(once)} \\
\Box_I\, \varphi &:= \neg \Diamond \neg\varphi \quad \textit{(always)} & \blacksquare_I\, \varphi &:= \neg \blacklozenge_I \neg\varphi \quad \textit{(historically)}
\end{aligned}
$$

Definition 2.22 *The* Language *of a MFOTL formula φ is*

$$\mathcal{L}(\varphi) = \{\sigma \in \mathbb{T}_\omega \mid \exists v.v, 0 \vDash \sigma\}$$

Example 2.23 *An example of an MFOTL formula over the signature Σ from Example 2.2 is*

$$
\begin{aligned}
\varphi := \Box_{[0,\infty)} (&\mathsf{sendEmail}(\mathsf{user}, \textit{"marketing"}) \rightarrow \\
&(\neg\mathsf{revokeEmailConsent}(\mathsf{user}, \textit{"marketing"})) \\
&\quad \mathsf{S}_{[0,\infty)}\ \mathsf{giveEmailConsent}(\mathsf{user}, \textit{"marketing"}))
\end{aligned}
$$

In natural language the policy in Example 2.23 requires that sending a marketing email requires that the receiving user has previously consented to receiving marketing emails and has not revoked this consent.

2.2 Runtime Verification

Runtime Verification is a research domain that focuses on the behavior of a system under scrutiny (SuS) in relation to a specific policy. Runtime verification can be subdivided into two related techniques: Runtime Monitoring and Runtime Enforcement.

Runtime Monitoring is used to detect a policy violation either in an online (real-time) or offline fashion. In runtime monitoring a monitor continuously evaluates the current and past system state and determines whether it complies with some policy. For our purposes the system state gets abstracted into a trace of events and the policy represents a set of allowed traces. Since the monitor never interacts with the system itself, runtime monitoring can also be done after a program has finished executing.

Runtime Enforcement is used to actively cause or prevent certain system behaviors. Due to this interactive nature of runtime enforcement it is inherently online. If we consider our trace model on a high level, an enforcer can "edit" any time-point before it gets fully committed to the trace and additionally the enforcer can create its own time-points at any increment of time. The following subsection explores an approach for enforcing MFOTL policies.

Figure 2.3: Proactive real-time first-order enforcement system model [9, Figure 2]

2.2.1 Proactive Runtime Enforcement using MFOTL

We consider the system model of Hublet et al. [9] which is shown in Figure 2.3. The system (under scrutiny) S runs in some environment X. The environment X is not under the control of the enforcer E. The enforcer E must ensure that the system S complies with some policy P. To this end the enforcer is given two main ways of interaction with the system. First, it can react to observable events happening in the system via issuing R-commands (reactively) by causing events of its own and/or suppressing events before they are visible to the environment. And second, the enforcer can issue P-commands (proactively) by causing certain events at some time-stamp τ.

In the following the event names of events in sets denoted by D_C are all assumed to be in $\mathbb{E}_\mathsf{C}$ and D_S are all assumed to be in $\mathbb{E}_\mathsf{S}$. The next three definitions reintroduce the three types of commands used by Hublet et al. [9].

Definition 2.24 *An* R-Command *has the form* $\mathsf{RCom}\,(D_\mathsf{C}, D_\mathsf{S})$*. This command is a reaction of the enforcer to some set of events D, that the system sent to the enforcer. The trace of the enforced system will contain the set $D_\mathsf{C} \cup (D \setminus D_\mathsf{S})$ of events.*

Definition 2.25 *A* P-Command *has the form* $\mathsf{PCom}\,(D_\mathsf{C})$*. This command can be proactively issued by the enforcer before any clock tick to issue the set of events D_C.*

Definition 2.26 No-Command *has the form* NoCom *and indicates that the enforcer takes no action.*

Definition 2.27 *The set of all commands is $\mathcal{C}$.*

This leads us to Hublet et al.'s [9, 10] definition of a proactive enforcer.

Definition 2.28 *An* Enforcer $\mathcal{E}$ *is a triple $(\mathcal{S}, s_0, \mu)$, where $\mathcal{S}$ is a set of enforcer states $s_0 \in \mathcal{S}$ is an initial enforcer state and $\mu : \mathbb{T}_f \times \mathcal{S} \times \mathbb{N} \times (\mathsf{DB} \cup \{\mathsf{tick}\}) \to \mathcal{C} \times \mathcal{S}$ is a computable update function such that the following two conditions hold*

1. $\forall \sigma \in \mathbb{T}_f, \tau \in \mathbb{N}, D \in \mathsf{DB}, s \in \mathcal{S}.\ \exists D_\mathsf{C} \in \mathsf{DB}_\mathsf{C}, D_\mathsf{S} \in \mathsf{DB}_\mathsf{S}, s' \in \mathcal{S}.$

$$\mu(\sigma, s, \tau, D) = (\mathsf{RCom}(D_\mathsf{C}, D_\mathsf{S}), s') \wedge D_\mathsf{C} \subseteq D$$

2. $\forall \sigma \in \mathbb{T}_f, s \in \mathcal{S}, \tau \in \mathbb{N}.\ \exists D_\mathsf{C} \in \mathsf{DB}_\mathsf{C}, s' \in \mathcal{S}.$

$$\mu(\sigma, s, \tau, \mathsf{tick}) \in \{(\mathsf{PCom}(D_\mathsf{C}), s'), (\mathsf{NoCom}, s')\}$$

Algorithm 1 Enforced trace [10, Fig. 2]

$\mathsf{run}(s, \sigma, \sigma', \tau) = \textbf{case } \sigma' \textbf{ of}$
$\mid \varepsilon \Rightarrow \varepsilon$
$\mid (\tau', D) \cdot \sigma'' \textbf{ when } \tau' > \tau \Rightarrow \textbf{let } (o, s') = \mu(\sigma, s, \tau, \mathsf{tick}) \textbf{ in}$
$\quad \textbf{case } o \textbf{ of} \mid \mathsf{PCom}(D_{\mathbb{C}}) \Rightarrow (\tau, D_{\mathbb{C}}) \cdot \mathsf{run}(s', \sigma \cdot (\tau, D_{\mathbb{C}}), \sigma', \tau + 1)$
$\qquad\qquad \mid \mathsf{NoCom} \Rightarrow \mathsf{run}(s', \sigma, \sigma', \tau + 1)$
$\mid (\tau', D) \cdot \sigma'' \textbf{ when } \tau' = \tau \Rightarrow \textbf{let } (o, s') = \mu(\sigma, s, \tau, D); D' = (D \setminus D_{\mathbb{S}}) \cup D_{\mathbb{C}}) \textbf{ in}$
$\quad \textbf{case } o \textbf{ of} \mid \mathsf{RCom}(D_{\mathbb{C}}, D_{\mathbb{S}}) \Rightarrow (\tau, D') \cdot \mathsf{run}(s', \sigma \cdot (\tau, D'), \sigma'', \tau + 1)$
$\mathcal{E}(\sigma) = \mathsf{run}(s_0, \varepsilon, \sigma, \textbf{if } \sigma = \varepsilon \textbf{ then } 0 \textbf{ else } \mathsf{fts}(\sigma))$

Hublet et al. [10] define an enforced trace algorithmically via Algorithm 1.

Definition 2.29 *For any $\sigma \in \mathbb{T}$ and enforcer $\mathcal{E} = (\mathcal{S}, s_0, \mu)$, the* enforced trace *$\mathcal{E}(\sigma)$ is defined co-recursively in Algorithm 1.*

In Definition 2.29 the trace σ corresponds to a system's behavior if it was running completely without an enforcer. The enforced trace then represents the systems behavior combined with those of the enforcer.

Definition 2.30 *An enforcer $\mathcal{E}$ is* sound *with respect to some property P, iff for any trace $\sigma \in \mathbb{T}_\omega$, $\mathcal{E}(\sigma) \in P$.*

Definition 2.31 *An enforcer $\mathcal{E}$ is* sound *with respect to some policy φ, iff it is sound for the language $\mathcal{L}(\varphi)$.*

Soundness allows us to describe and capture the behavior of an enforcer.

Definition 2.32 *An enforcer $\mathcal{E} = (\mathcal{S}, s_0, \mu)$ is transparent with respect to some property P iff for all $\sigma \in P$, $\mathcal{E}(\sigma) = \sigma$.*

Definition 2.33 *An enforcer $\mathcal{E} = (\mathcal{S}, s_0, \mu)$ is transparent with respect to some policy φ iff it is transparent for the language $\mathcal{L}(\varphi)$.*

Transparency means that an enforcer will not alter a trace that is already compliant with some policy or property. A property or policy is said to be *(transparently) enforceable* if there exists a (transparent) sound enforcer for the policy or property.

2.2.2 Enforceable MFOTL

Capabilities are an integral part of the proactive enforcement approach introduced by Hublet et. al [9]. Under this approach an enforcer can have different capabilities, or ways of interacting with events emitted by a given system under scrutiny (SuS). In Hublet et. al the set of capabilities consists of observable ($\mathbb{O}$), suppressable ($\mathbb{S}$), causable ($\mathbb{C}$), and suppressable-causable

$$\begin{tikzcd}
 & & \mathbb{C}\mathbb{S} & \\
 & \mathbb{C}\mathbb{O} \arrow[ur] & & \mathbb{S} \arrow[ul] \\
\mathbb{C} \arrow[ur] & & \mathbb{O} \arrow[ul] \arrow[ur] & \\
 & \bot \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

Figure 2.4: The *Capability Lattice* $(\mathbb{K}, \preceq)$

($\mathbb{C}\mathbb{S}$). Hublet et. al [12] expands this expands this system of capabilities, specifically it makes the existence of events that may not (always) be externally visible explicit. Under both notions a capability function $\kappa : \mathbb{E} \to \mathbb{K}$ is used to assign a capability to all events with some event name $e \in \mathbb{E}$. Also in both contexts, capabilities form a (semi)-lattice $(\mathbb{K}, \preceq)$.

Definition 2.34 *The subset of* Causable Events

$$\mathbb{E}_{\mathbb{C}} = \{e \mid \mathbb{C} \preceq \kappa(e)\}$$

consists of all event names which can be caused by the enforcer, i.e. which have a capability of at least $\mathbb{C}$*.*

Definition 2.35 *The subset of* Suppressable Events

$$\mathbb{E}_{\mathbb{S}} = \{e \mid \mathbb{S} \preceq \kappa(e)\}$$

consists of all event names which can be suppressed by the enforcer, i.e. which have a capability of at least $\mathbb{S}$*.*

Definition 2.36 *The set of* Causable Databases $\mathsf{DB}_{\mathbb{C}}$ *contains all sets of events that are at least causable.*

$$\mathsf{DB}_{\mathbb{C}} = \{D \in \mathsf{DB} \mid \forall e\,(\bar{t}) \in D.\ e \in \mathbb{E}_{\mathbb{C}}\}$$

Definition 2.37 *The set of* Suppressable Databases $\mathsf{DB}_{\mathbb{S}}$ *contains all sets of events that are at least suppressable.*

$$\mathsf{DB}_{\mathbb{S}} = \{D \in \mathsf{DB} \mid \forall e\,(\bar{t}) \in D.\ e \in \mathbb{E}_{\mathbb{S}}\}$$

On the basis of capabilities Hublet, Lima et al. [9, 10] introduce a subset of MFOTL formulae that is enforceable. The typing rules for this subset are shown in Figure 2.6. These rules rely on the concept of past-guardedness, specifically for the suppressability of existential quantifiers. Past-guardedness means that all possible valuations of a variable in the past are known. The rules for past-guardedness are shown in Figure 2.5.

$$\frac{}{\gamma \vdash e(\dots, x, \dots) : \mathsf{PG}_E(x)^+}\,\mathbb{E}^+_{\mathsf{PG}} \qquad \frac{\gamma \vdash \varphi : \mathsf{PG}_E(x)^{\neg p}}{\gamma \vdash \neg\varphi : \mathsf{PG}_E(x)^p}\,\neg_{\mathsf{PG}}$$

$$\frac{x \neq z \quad \gamma \vdash \varphi : \mathsf{PG}_E(z)^p}{\gamma \vdash \exists x.\, \varphi : \mathsf{PG}_E(z)^p}\,\exists_{\mathsf{PG}} \qquad \frac{\gamma \vdash \varphi : \mathsf{PG}_E(x)^+}{\gamma \vdash \varphi \wedge \psi : \mathsf{PG}_E(x)^+}\,\wedge^{\mathsf{L}+}_{\mathsf{PG}}$$

$$\frac{\gamma \vdash \psi : \mathsf{PG}_E(x)^+}{\gamma \vdash \varphi \wedge \psi : \mathsf{PG}_E(x)^+}\,\wedge^{\mathsf{R}+}_{\mathsf{PG}} \qquad \frac{\gamma \vdash \varphi : \mathsf{PG}_E(x)^- \quad \gamma \vdash \psi : \mathsf{PG}_E(x)^-}{\gamma \vdash \varphi \wedge \psi : \mathsf{PG}_E(x)^-}\,\wedge^{-}_{\mathsf{PG}}$$

$$\frac{0 \notin I \quad \gamma \vdash \varphi : \mathsf{PG}_E(x)^+}{\gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathsf{PG}_E(x)^+}\,\mathsf{S}^{\mathsf{L}+}_{\mathsf{PG}} \qquad \frac{\gamma \vdash \psi : \mathsf{PG}_E(x)^+}{\gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathsf{PG}_E(x)^+}\,\mathsf{S}^{\mathsf{R}+}_{\mathsf{PG}}$$

$$\frac{0 \in I \quad \gamma \vdash \psi : \mathsf{PG}_E(x)^-}{\gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathsf{PG}_E(x)^-}\,\mathsf{S}^{-}_{\mathsf{PG}} \qquad \frac{0 \notin I \quad \gamma \vdash \varphi : \mathsf{PG}_E(x)^+}{\gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathsf{PG}_E(x)^+}\,\mathsf{U}^{\mathsf{L}+}_{\mathsf{PG}}$$

$$\frac{\gamma \vdash \varphi : \mathsf{PG}_E(x)^+ \quad \gamma \vdash \psi : \mathsf{PG}_E(x)^+}{\gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathsf{PG}(_E x)^+}\,\mathsf{U}^{\mathsf{LR}+}_{\mathsf{PG}} \qquad \frac{0 \in I \quad \gamma \vdash \psi : \mathsf{PG}_E(x)^-}{\gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathsf{PG}_E(x)^-}\,\mathsf{U}^{-}_{\mathsf{PG}}$$

$$\frac{\gamma \vdash \varphi : \mathsf{PG}_E(x)^+}{\gamma \vdash \bullet_I\, \varphi : \mathsf{PG}_E(x)^+}\,\bullet^+_{\mathsf{PG}} \qquad \frac{\gamma((e,i)) = E}{\gamma \vdash e(x_1, \dots, x_i = x, \dots, x_k) : \mathsf{PG}^p_E(x)}\,\gamma_{\mathsf{PG}}$$

$$\frac{\forall i.\ \gamma \vdash \varphi_1 : \mathsf{PG}^p_{E_i}(x_i) \quad \gamma, (e,1) : E_1, \dots, (e,k) : E_k \vdash \varphi_2 : \mathsf{PG}^p_E(x)}{\gamma \vdash \mathsf{let}\, e(x_1, \dots, x_k) = \varphi_1 \,\mathsf{in}\, \varphi_2 : \mathsf{PG}^p_E(x)}\,\mathsf{let}_{\mathsf{PG}}$$

Figure 2.5: Past-Guardedness Rules

The signature Σ of Example 2.2 could be extended with the following capabilities:

$$\begin{aligned} \kappa(\mathsf{sendEmail}) &= \mathbb{CS} \\ \kappa(\mathsf{giveEmailConsent}) &= \mathbb{O} \\ \kappa(\mathsf{revokeEmailConsent}) &= \mathbb{O}. \end{aligned}$$

These capabilities capture, that an enforcer can both send and prevent the sending of emails, but it cannot change the consent given or revoked by a user.

2.2.3 Constitutive, Regulative, and Exception Rules

2.2.4 Introduction

Hublet et. al [12] introduce a meta-logic based on constitutive, regulative, and exception rules (CRE meta-logic). The CRE meta-logic underpins the Lex programming language and formalizes how a Lex specification would compiles to an MFOTL formula.

Definition 2.38 *The* set of rule labels *is denoted by* $\mathcal{R}$.

$$\frac{}{\Gamma \vdash \top : \mathbb{C}}\ \top^{\mathbb{C}} \qquad \frac{}{\Gamma \vdash \bot : \mathbb{S}}\ \bot^{\mathbb{S}} \qquad \frac{e \in \mathbb{C} \quad \Gamma(e) = \mathbb{C}}{\Gamma \vdash e(t_1, \dots, t_k) : \mathbb{C}}\ \mathbb{E}^{\mathbb{C}}$$

$$\frac{e \in \mathbb{S} \quad \Gamma(e) = \mathbb{S}}{\Gamma \vdash e(t_1, \dots, t_k) : \mathbb{S}}\ \mathbb{E}^{\mathbb{S}} \qquad \frac{\Gamma \vdash \varphi : \mathbb{S}}{\Gamma \vdash \neg\varphi : \mathbb{C}}\ \neg^{\mathbb{C}} \qquad \frac{\Gamma \vdash \varphi : \mathbb{C}}{\Gamma \vdash \neg\varphi : \mathbb{S}}\ \neg^{\mathbb{S}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{C}}{\Gamma \vdash \exists x.\ \varphi : \mathbb{C}}\ \exists^{\mathbb{C}} \qquad \frac{\Gamma \vdash \varphi : \mathbb{S} \quad \vdash \varphi : \mathsf{PG}(x)^{+}}{\Gamma \vdash \exists x.\ \varphi : \mathbb{S}}\ \exists^{\mathbb{S}} \qquad \frac{\Gamma \vdash \varphi : \mathbb{C} \quad \Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi \wedge \psi : \mathbb{C}}\ \wedge^{\mathbb{C}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{S}}{\Gamma \vdash \varphi \wedge \psi : \mathbb{S}}\ \wedge^{\mathsf{SL}} \qquad \frac{\Gamma \vdash \psi : \mathbb{S}}{\Gamma \vdash \varphi \wedge \psi : \mathbb{S}}\ \wedge^{\mathsf{SR}} \qquad \frac{0 \in I \quad \Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{C}}\ \mathsf{S}^{\mathsf{C}}$$

$$\frac{0 \notin I \quad \Gamma \vdash \varphi : \mathbb{S}}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{S}}\ \mathsf{S}^{\mathsf{SL}} \qquad \frac{0 \in I \quad \Gamma \vdash \varphi : \mathbb{S} \quad \Gamma \vdash \psi : \mathbb{S}}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{S}}\ \mathsf{S}^{\mathsf{SLR}}$$

$$\frac{\Gamma \vdash \psi : \mathbb{S}}{\Gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathbb{S}}\ \mathsf{U}^{\mathsf{S}} \qquad \frac{b \neq \infty \quad \Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi\, \mathsf{U}_{[0,b]}\, \psi : \mathbb{C}}\ \mathsf{U}^{\mathsf{CR}}$$

$$\frac{b \neq \infty \quad \Gamma \vdash \varphi : \mathbb{C} \quad \Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi\, \mathsf{U}_{[a,b]}\, \psi : \mathbb{C}}\ \mathsf{U}^{\mathsf{CLR}} \qquad \frac{\Gamma \vdash \varphi : \mathbb{C} \quad b > 0}{\Gamma \vdash \bigcirc_{[0,b)}\, \varphi : \mathbb{C}}\ \bigcirc^{\mathsf{C}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{S}}{\Gamma \vdash \bigcirc_I\, \varphi : \mathbb{S}}\ \bigcirc^{\mathsf{S}} \qquad \frac{\Gamma \vdash \varphi_1 : \tau_1 \quad \Gamma, e : \tau_1 \vdash \varphi_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, e(\bar{t}) = \varphi_1\ \mathsf{in}\ \varphi_2 : \tau_2}\ \mathsf{let}$$

Figure 2.6: EMFOTL Typing Rules

Definition 2.39 *A* CRE[MFOTL] Signature *is a quadruple of the following form*

$$\Sigma' = (\mathbb{D}, \mathbb{E} \cup \mathbb{E}', \mathbb{F}, \iota \cup \iota').$$

Where $\mathbb{D}$ is a domain of values, $\mathbb{E}$ is a set of external event names, $\mathbb{E}'$ is a set of internal event names, $\mathbb{F}$ is a set of function names, $\iota : \mathbb{E} \cup \mathbb{F} \to \mathbb{N}$ is an external arity funciton, and $\iota' : \mathbb{E}' \to \mathbb{N}$ is an internal arity function. $\mathbb{E}$ and $\mathbb{E}'$ are disjoint.

CRE[MFOTL] signatures are very similar to MFOTL signatures, but include the addition of internal events. While it is useful to distinguish between internal and external events, in CRE[MFOTL] and later in Lex, a CRE[MFOTL] signature can still be used as an MFOTL signature. Therefore the next definition deifnes the set of valid MFOTL formulae over a CRE[MFOTL] signature.

Definition 2.40 *The* set of all MFOTL formulae *over a CRE[MFOTL] signature Σ' is denoted by $\mathcal{F}$.*

Next we recall constitutive, regulative, and exception rules [12, Section 3.1].

Definition 2.41 *Let $r, s \in \mathcal{R}$, $\varphi, \psi \in \mathcal{F}$, $e \in \mathbb{E}'$, $x_1, \dots, x_{\iota'(e)} \in \mathbb{D}$, and $\bar{x} = (x_1, \dots, x_{\iota'(e)})$. Table 2.1 gives an overview over the three rule types, their syntax, and their meaning.*

Rule Type	Syntax	Meaning
Constitutive	$r : \varphi \Rightarrow e(\bar{x})$	"φ counts as $e(\bar{x})$"
Regulative	$r : \varphi \Rightarrow \psi$	"if φ holds, then ψ must hold"
Exception	$r : \varphi \Rightarrow \mathsf{exc}(s)$	"if φ holds, then rule s shall not apply"

Table 2.1: CRE[MFOTL] Rule Types

$$
\begin{array}{lll}
[\textsc{con}] & \gamma \models^{\Phi} p & \text{iff } r : \varphi \Rightarrow p \in C \text{ and } \gamma \models^{\Phi} \varphi \wedge \neg\mathsf{exc}(r) \\
[\textsc{reg}] & \gamma \models^{\Phi} \mathsf{comply}(r) & \text{iff } r : \varphi \Rightarrow \psi \in R \text{ and } \gamma \models^{\Phi} (\varphi \rightarrow \psi) \vee \mathsf{exc}(r) \\
[\textsc{exc}] & \gamma \models^{\Phi} \mathsf{exc}(s) & \text{iff } r : \varphi \Rightarrow \mathsf{exc}(s) \in E \text{ and } \gamma \models^{\Phi} \varphi \wedge \neg\mathsf{exc}(r)
\end{array}
$$

Figure 2.7: Semantics of CRE for a fixed structure [12, Figure 5]

Definition 2.42 *A* CRE Policy *is a triple $\Phi = (C, R, E)$ where C is a set of constitutive rules, R is a set of regulative rules, and E is a set of exception rules such that every rule in $C \cup R \cup E$ has a unique label.*

Figure 2.7 defines the semantics of the CRE[MFOTL] logic. The semantics expand the logic of $\mathcal{F}$ the CRE[MFOTL] logic is captured by $\gamma \models \varphi$. These semantics can be expanded to define compliance for an entire CRE-policy as followed.

Definition 2.43 *A CRE-policy $\Phi = (C, R, R)$ is complied with ($\gamma \models \Phi$), if all its regulative rules are complied with, which is captured in the rule:*

$$
\frac{\forall r : \varphi \Rightarrow \psi \in R.\ \mathsf{comply}(r)}{\gamma \models \Phi}\ \textit{CRE-Compliance}
$$

Definition 2.44 *A* CRE Specification *$\mathcal{S} = (\Sigma, \varphi, \kappa)$ is a triple consisting of some signature $\Sigma = (\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota)$, some CRE policy $\Phi = (C, R, E)$, and a capability-mapping κ.*

Analogous to EMFOTL, Hublet et al. [12] define enforceable CRE (ECRE) via the typing rules in Figure 2.8

Example 2.45 *The MFOTL policy Φ of Example 2.23 could be captured in the following CRE[MFOTL] policy:*

$$
\begin{aligned}
C = \{ & r : \mathsf{sendEmail}(\mathsf{user}, \textit{"marketing"}) \Rightarrow \\
& \quad (\neg\mathsf{revokeEmailConsent}(\mathsf{user}, \textit{"marketing"})) \\
& \qquad \mathsf{S}_{[0,\infty)}\ \mathsf{giveEmailConsent}(\mathsf{user}, \textit{"marketing"})\} \\
R = {} & \emptyset \\
E = {} & \emptyset
\end{aligned}
$$

$$\frac{r : \varphi \Rightarrow \psi \in R \quad \overline{x} = \mathsf{fv}(\varphi) \cup \mathsf{fv}(\psi) \quad \Gamma \vdash^{\Phi} \Box(\forall \overline{x}.\ \varphi \to \psi) : \mathbb{C}}{\Gamma \vdash^{\Phi} \mathsf{comply}(r) : \mathbb{C}}\ \textsc{reg}\mathrm{I}$$

$$\frac{r : \varphi \Rightarrow \psi \in R \quad \overline{x} = \mathsf{fv}(\varphi) \cup \mathsf{fv}(\psi) \quad \Gamma \vdash^{\Phi} \Box(\forall \overline{x}.\ \varphi \land \neg\psi \to \mathsf{exc}(r)) : \mathbb{C}}{\Gamma \vdash^{\Phi} \mathsf{comply}(r) : \mathbb{C}}\ \textsc{reg}\mathrm{E}$$

$$\frac{r : \varphi \Rightarrow \mathsf{exc}(s) \in E \quad \Gamma \vdash^{\Phi} \varphi : \mathbb{C} \quad \Gamma \vdash^{\Phi} \mathsf{exc}(r) : \mathbb{S}}{\Gamma \vdash^{\Phi} \mathsf{exc}(s) : \mathbb{C}}\ \textsc{exc}^{\mathbb{C}}$$

$$\frac{\forall r : \varphi \Rightarrow \mathsf{exc}(s) \in E.\ (\Gamma \vdash^{\Phi} \varphi : \mathbb{S} \textbf{ or } \Gamma \vdash^{\Phi} \mathsf{exc}(r) : \mathbb{C})}{\Gamma \vdash^{\Phi} \mathsf{exc}(s) : \mathbb{S}}\ \textsc{exc}^{\mathbb{S}}$$

$$\frac{r : \varphi \Rightarrow p \in C \quad \Gamma \vdash^{\Phi} \varphi : \mathbb{C} \quad \Gamma \vdash^{\Phi} \mathsf{exc}(r) : \mathbb{S}}{\Gamma \vdash^{\Phi} p : \mathbb{C}}\ \textsc{con}^{\mathbb{C}}$$

$$\frac{\forall r : \varphi \Rightarrow p \in C.\ (\Gamma \vdash^{\Phi} \varphi : \mathbb{S} \textbf{ or } \Gamma \vdash^{\Phi} \mathsf{exc}(r) : \mathbb{C})}{\Gamma \vdash^{\Phi} p : \mathbb{S}}\ \textsc{con}^{\mathbb{S}}$$

Figure 2.8: Typing rules for ECRE[MFOTL] policies [12, Figure 6]

$\Phi := (C, R, E)$ *is a CRE[MFOTL] policy consisting of a single regulative rule* r*.*

2.2.5 Compilation of CRE[MFOTL] to MFOTL

This section recalls the compilation algorithm of CRE[MFOTL] to Lex as it is introduced in Hublet et al. [12, Section 3.2]. Hublet et al. [12] define the following dependency relation between constitutive and exception rules analogous to Merigoux et al. [15, Section 4.1].

Definition 2.46 *For a CRE[MFOTL] Policy* $\Phi := (C, R, E)$ *let* $r, s \in C \cup E$*.* $r \rhd_{\Phi} s$ *(read as "s depends on r") iff either* $r : \varphi \Rightarrow p \in C$ *and* p *occurs in the left-hand side of the rule* s*, or* $r : \varphi \Rightarrow \mathsf{exc}(s) \in E$*.*

A policy (C, R, E) is called *acyclic* iff there do not exist two rules $r, s \in C \cup E$ such that $r \rhd_{\Phi} s$, and $s \rhd_{\Phi} r$ and also that there is no single rule $r \in C \cup E$ that is self-referential (i.e. $r \rhd_{\Phi} r$). The compilation of CRE[MFOTL] to MFOTL is defined by Algorithm 2 and Algorithm 3.

2.3 The Lex Programming Language

Hublet et al. [12] and Degelo [8] developed the Lex programming language based on CRE[MFOTL]. Lex is designed to formalize legal texts and maintain a high readability, especially compared to pure MFOTL. It achieves this readability by maintaining the general structure of the underlying legal text.

Algorithm 2 CRE[MFOTL] to MFOTL compilation [12, Algorithm 1]

Input: $\mathcal{S} = (\Sigma, \Phi, \kappa)$ with $\Sigma = (\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota)$ and acyclic CRE[MFOTL] policy $\Phi = (C, R, E)$
$\mathbb{E}_E \leftarrow \{e_s \mid r : \varphi \Rightarrow \mathsf{exc}(s) \in E\}$
$R' \leftarrow \{r : \varphi \wedge \neg e_r \Rightarrow \psi \mid r : \varphi \Rightarrow \psi \in R, e_r \in \mathbb{E}_E\} \cup \{r : \varphi \Rightarrow \psi \mid r : \varphi \Rightarrow \psi \in R, e_r \notin \mathbb{E}_E\}$
$C_E \leftarrow \{r : \varphi \Rightarrow e_s \mid r : \varphi \Rightarrow \mathsf{exc}(s) \in E\}$
$\mathbb{E}_0 \leftarrow \{p \mid p \notin \mathbb{E} \wedge e \text{ occurs in } C \cup R \cup E\} \setminus \{p \mid r : \varphi \Rightarrow p \in C\}$
$C_0 \leftarrow \{z_p : \bot \Rightarrow p \mid p \in \mathbb{E}_0\}$
$C' \leftarrow C \cup C_E \cup C_0$
$C' \leftarrow \{r : \varphi \wedge \neg e_r \Rightarrow p \mid r : \varphi \Rightarrow p \in C', e_r \in \mathbb{E}_E\} \cup \{r : \varphi \Rightarrow p \mid r : \varphi \Rightarrow p \in R, e_r \notin \mathbb{E}_E\}$
$[(\varphi_1, p_1), \ldots, (\varphi_k, p_k)] \leftarrow \textsc{collect}(\textsc{topological\_sort}_{\triangleright_\Phi}(C'))$
for $r : \varphi \Rightarrow \psi \in R'$ **do** $\overline{x}_r \leftarrow \mathsf{free\_variables}(\varphi \to \psi)$ **endfor**
$\varphi \leftarrow \mathsf{let}\ p_1 = \varphi_1\ \mathsf{in} \ldots \mathsf{let}\ p_k = \varphi_k\ \mathsf{in} \bigwedge_{r:\varphi\Rightarrow\psi\in R'} (\Box(\forall \overline{x}_r.\ \varphi \to \psi))$
Output: $\mathcal{S}' = (\Sigma', \varphi, \kappa)$ where $\Sigma' = (\mathbb{D}, \mathbb{E} \setminus \{p_i \mid 1 \le i \le k\}, \mathbb{F}, \iota)$

Algorithm 3 Auxiliary function COLLECT [12, Algorithm 2]

let collect L =
  $P, L' \leftarrow \emptyset, []$
  for $r : \varphi \Rightarrow p \in L$ **do**
    if $p \notin P$ **then** $(P, L') \leftarrow (P \cup \{p\}, L' \cdot [(\bigvee_{r:\varphi\Rightarrow p\in L} \varphi, p)])$ **endif**
  endfor
  return L'

2.3.1 Lex Syntax

Lex has the following main types of statements:

- **Section Labels** (keywords: `law`, `title`, `chapter`, `section`, `article`, `paragraph`, `point`, `subpoint`) Section labels are used to represent the structure of the law. Section label statements have the form of a section keyword followed by a mandatory string identifier and an optional, more descriptive, string title (e.g. `article` "1" "Preconditions"). Section labels (together with optional rule identifiers) are the analog of rule labels in CRE[MFOTL]. Section 2.3.2 will go more in depth on section labels in Lex.

- **Type Declarations** (keyword: `type`) Type declarations map a type alias to a base type (e.g. `type` user `is` string, `type` age `is` int). These type aliases are not represented in CRE[MFOTL].

- **Event Declarations** (keywords: `event`, `predicate`) Event declarations declare event names and their arguments along with their types.

`Lex`	`law` *lbl* `["title"]`
	`article` *lbl* `["title"]`
	`paragraph` *lbl*...
	`point` *lbl*...
	...
	
	`type` t_1 `[is` t_2`]`
	`"""<docstring>"""`
		$[\tau]$ `event` e
		`"""<docstring>"""`
		$x_1 : t_1$
		...
		$x_k : t_k$
		$\tau ::=$ `observable`
		$\mid$ `suppressable`
		$\mid$ `causable` $\mid$ `internal`...

CRE[MFOTL]	n.a.	$e \in \mathbb{E}$ $\iota(e) = (t_1, \ldots, t_k)$ $\kappa(e) = \mathbb{O}, \mathbb{S}, \mathbb{C}\ldots$

`rule` r
`whenever`
φ_1
...
φ_k
`constitute`
p_1
...
p_ℓ
	`rule` r
	`whenever`
	φ_1
	...
	φ_k
	`oblige`
	ψ_1
	...
	ψ_ℓ
		`rule` r
		`whenever`
		φ_1
		...
		φ_k
		`except`
		lbl_1
		...
		lbl_ℓ

$r_1 : \bigwedge_{1 \leq i \leq k} \varphi_i \Rightarrow p_1 \in C$ ... $r_k : \bigwedge_{1 \leq i \leq k} \varphi_\ell \Rightarrow p_\ell \in C$	$r_1 : \bigwedge_{1 \leq i \leq k} \varphi_i \Rightarrow \psi_1 \in R$ ... $r_k : \bigwedge_{1 \leq i \leq k} \varphi_\ell \Rightarrow \psi_\ell \in R$	$r_1 : \bigwedge_{1 \leq i \leq k} \varphi_i \Rightarrow \mathsf{exc}(s_1) \in E$ ... $r_k : \bigwedge_{1 \leq i \leq k} \varphi_\ell \Rightarrow \mathsf{exc}(s_\ell) \in E$

Figure 2.9: Correspondence between the `Lex` and CRE[MFOTL] syntax [12, Figure 10]

- **Function Declarations** (keyword: `function`) Function declarations are analogous to event declarations, but they define function names with their arguments and a return type.
- **Rule Definitions** (keyword: `rule`) Rule definitions are the core of the `Lex` language. They define the constitutive, regulative, and exception rules like in CRE[MFOTL]. `Lex` rules are more verbose than those in CRE[MFOTL] and allow for the use of keywords in addition to symbols for temporal/logical operators (e.g. `ALWAYS` for $\square$). A CRE[MFOTL] rule like $r : \varphi \Rightarrow p \in C$ becomse `rule whenever` φ `constitute` p in `Lex`. After the `rule` keyword, a rule can optionally be given a string identifier. Exception rules in `Lex` make use of section labels and the optional rule identifiers to specify which rule(s) the apply to.
- **Import Statements** (keyword: `import`) Import statements are used to modularly split a `Lex` specification into multiple files. "`import` specifi-

cation" imports the specification in the file "specification.lex".

Figure 2.9 shows the correspondence between Lex and CRE[MFOTL].

2.3.2 Lex Section Labels

The Lex language provides a fixed set of section label keywords: `law`, `title`, `chapter`, `section`, `article`, `paragraph`, `point`, `subpoint`. Not every law will have all of these hierarchical levels, and besides `law` and `article` they are all optional. If a law contains a hierarchical level that does not fit into these predifined keywords, custom sub-levels can be defined underneath one of the predifined ones (e.g. `article`[1] is a level below an article, but above a paragraph; This has the consequence that `article` is implicitly equivalent to `article`[0]). As an example, a law might be the GDPR, which is further divided into chapters and chapters then contain individual articles. Another example would be the Swiss constitution as a law, which is divided into titles. While often times sections, titles, articles, etc. will be numbered, in Lex they are refered to by strings. This is due to the fact that no standard numbering scheme exists, sometimes articles will contain letters like Article 1b, and other times Roman numerals might be used. One interesting aspect of laws across contexts is that articles tend to be numbered continuously, i.e. in the GDPR one doesn't need to know in which Chapter an article is located, to uniquely refer to it.

References to other parts of a Lex specification can always be made by stating all levels can always be made by giving the full path consisting of all intermediate levels. Due to the continuous (and thus unique) numbering of articles, for sections below the `article` level, it is not necessary to include any levels between law and article. Lex will also try to resolve references dynamically, e.g. if `article` "2" is used in `article` "1", Lex will attempt to resolve this reference inside the same `law`.

2.3.3 Lex - E-Mail Marketing Example

This subsection will showcase the Lex language on a hands on example. Chapter 4 will continue this example to showcase how a Lex specification can be refined. The example is based on the ePrivacy Directive of the European Union. Specifically we focus on Article 13 of the ePrivacy Directive. This article regulates the use of electronic communication, automated calls, fax, and e-mail for marketing purposes.

Figure 2.11 shows one way that Article 13 could be formalized in Lex. It defines three data types, `person`, `purpose`, and `productCategory` to represent the involved parties: `person` to represent both legal and natural persons, `purpose` to represent the purpose of the communication, and `productCategory` to represent the category of the product or service.

*Art. 13 ePrivacy Directive . **Unsolicited communications***

1. *The use of automated calling systems without human intervention (automatic calling machines), facsimile machines (fax) or electronic mail for the purposes of direct marketing may only be allowed in respect of subscribers who have given their prior consent.*
2. *Notwithstanding paragraph 1, where a natural or legal person obtains from its customers their electronic contact details for electronic mail, in the context of the sale of a product or a service, in accordance with Directive 95/46/EC, the same natural or legal person may use these electronic contact details for direct marketing of its own similar products or services provided that customers clearly and distinctly are given the opportunity to object, free of charge and in an easy manner, to such use of electronic contact details when they are collected and on the occasion of each message in case the customer has not initially refused such use.*
3. *Member States shall take appropriate measures to ensure that [...] unsolicited communications for purposes of direct marketing, in cases other than those referred to in paragraphs 1 and 2, are not allowed [...]*
4. *In any event, [...] disguising or concealing the identity of the sender [...] shall be prohibited.*
5. *Paragraphs 1 and 3 shall apply to subscribers who are natural persons. Member States shall also ensure, [...] that the legitimate interests of subscribers other than natural persons with regard to unsolicited communications are sufficiently protected.*

Figure 2.10: Text of Article 13 of the ePrivacy Directive [1]

Next it specifies one internal event for automated communication and three causable and suppressable events for the three different types of automated communication (automated calls, faxes, and e-mails), mentionend in the directive. Via three constitutive rules, these concrete events do then constitute the internal event `automatedCommunication`.

After this the logical structure of Article 13 is formalized into three rules. It requires some interpretation to determine which parts can and should be formalized. Paragraph 3 is fairly general and does not directly specify any concrete rule. Paragraph 4 does state a concrete rule ("one shall not disguise one's identity"), however it can be a reasonable assumption to leave this honesty requirement to the system itself and enforce it with an enforcer. Finally, paragraph 5 restricts the scope of Paragraphs 1 and 3 to be only applicable to natural persons. As Article 13 stems from an EU directive, national implementations may differ. These differences, though relevant, are beyond the scope of this thesis.

Therefore, we are left with paragraphs 1, 2, and 5. Paragraph 1 contains a simple obligation that a person, likely a business, can only send marketing communication to a user, if the user gave consent for receiving such com-

```
law "EPD"

type productCategory is string

internal event automatedCommunication
    """persso {s} sends automated
    commmunication to person {r} for
    purpose {p} about category {c}"""
    s: person
    r: person
    p: purpose
    c: productCategory

causable suppressable event automatedCall
    """person {s} calls person {r} for
    purpose {p} about category {c}"""
    r: person
    s: person
    p: purpose
    c: productCategory

causable suppressable event fax
    """persen {s} faxes person {r} for
    purpose {p} about category {c}"""
    r: person
    s: person
    p: purpose
    c: productCategory

causable suppressable event email
    """person {s} emails person {r} for
    purpose {p} about category {c}"""
    r: person
    s: person
    p: purpose
    c: productCategory

observable event obtainedEmailForCategory
    """person {r} obtained the email
    address of person {s} for category {c}"
    ""
    r: person
    s: person
    c: productCategory

observable event marketingConsent
    """person {s} gave prior consent for
    receiving marketing communication from
    person {r}"""
    r: person
    s: person

observable event
    objectToSimilarEmailMarketing
    """person {r} objected to receiving
    marketing emails from person {s} about
    category {c}"""
    r: person
    s: person
    c: productCategory
type person
type purpose is string

observable event isNaturalPerson
    """person {s} is a natural person"""
    s: person

rule "automated call communication"
    whenever
        automatedCall(s, r, p, c)
    constitute
        automatedCommunication(s, r, p, c)

rule "fax communication"
    whenever
        fax(s, r, p, c)
    constitute
        automatedCommunication(s, r, p, c)

rule "email communication"
    whenever
        email(s, r, p, c)
    constitute
        automatedCommunication(s, r, p, c)


article "13" "Unsolicited communications"
paragraph "1"

rule "marketing consent requirement"
    whenever
        automatedCommunication(s, r, "
    marketing", c)
    oblige
        marketingConsent(r, s)
    enforceable suppressing
    automatedCommunication

paragraph "2"

rule "marketing of similar products or
    services"
    whenever
        obtainedEmailForCategory(r, s, c)
        NOT objectToSimilarEmailMarketing(r
    , s, c)
    except
        article "13" paragraph "1"

paragraph "3"
paragraph "4"
paragraph "5"

rule
    whenever
        isNaturalPerson(r)
    scope
        article "13" paragraph "1"
        article "13" paragraph "3"
```

Figure 2.11: Lex formalization of Article 15(1), 15(2) and 15(5) of the ePrivacy Directive

munication. And paragraph 2 states an exception to paragraph 1, where the business does not need explicit consent, as long as the contact information was obtained for a similar product or service, the user has not objected to receiving such communication, and the marketing for a product or service belonging to the sender. Paragraph 5 can be neatly formalized with a scope rule. A scope rule could be converted into an exception by negating the conditions.

2.4 Event Systems

Sprenger et al. [17] defines event systems in the following way.

Definition 2.47 *An* Event System *is a labeled transition system*

$$\Xi = (\mathfrak{S}, \mathfrak{E}, \rightarrow)$$

consisting of a set of states $\mathfrak{S}$, a set of events $\mathfrak{E}$, and a transition relation $\rightarrow \subseteq \mathfrak{S} \times \mathfrak{E} \times \mathfrak{S}$.

$(s, e, s') \in \rightarrow$ can also be written as $s \xrightarrow{e} s'$. Sprenger et. al [17] extend the transition relation for finite traces of events τ. As we already use τ for timestamps and σ for MFOTL traces, we use σ for event traces and ε denotes an empty trace. The extension of the transition relation is explained in the following definition.

Definition 2.48 *The transition relation is inductively expanded for finite traces σ of events $\mathfrak{S}$ for all $s, s', s'' \in \mathfrak{S}$*

$$s \xrightarrow{\varepsilon} s \tag{2.1}$$

$$s \xrightarrow{\sigma \cdot \langle e \rangle} s' \qquad \text{whenever } s \xrightarrow{\sigma} s' \text{ and } s' \xrightarrow{e} s'' \tag{2.2}$$

Next we recall the trace definition for event systems from Sprenger et. al [17].

Definition 2.49 *The set of* Initial States *is a subset $\mathfrak{I} \subseteq \mathfrak{S}$.*

Definition 2.50 *Given a set of initial states $\mathfrak{I} \subseteq \mathfrak{S}$ a* Trace *of the event system $\Xi = (\mathfrak{S}, \mathfrak{E}, \rightarrow)$ starting in initial states $\mathfrak{I}$ is a trace σ of events such that $s \xrightarrow{\rho} s'$ for some initial state $s \in \mathfrak{I}$ and a reachable state $s' \in \mathfrak{S}$.*

The set of all traces of an event system Ξ starting in initial states $\mathfrak{I}$ is denoted by $\mathsf{traces}(\Xi, \mathfrak{I})$.

2.5 Refinement of Event Systems

In this section we recall refinement of event systems as defined in Sprenger et. al [17].

Definition 2.51 *Given two event systems $\Xi_i = (\mathfrak{S}_i, \mathfrak{E}_i, \rightarrow_i)$ with initial states $\mathfrak{I}_i$ for $i \in \{1,2\}$. $(\Xi_2, \mathfrak{I}_2)$ refines $(\Xi_1, \mathfrak{I}_1)$ modulo a mediator function $\pi : \mathfrak{E}_2 \rightarrow \mathfrak{E}_1$, written as $(\Xi_2, \mathfrak{I}_2) \sqsubseteq_\pi (\Xi_1, \mathfrak{I}_1)$ if there exists a simulation relation $R \subseteq \mathfrak{E} \times \mathfrak{E}$ such that*

1. *for all $s_2 \in \mathfrak{I}_2$ there exists some $s_1 \in \mathfrak{I}_1$ such that $(s_2, s_1) \in R$, and*
2. *for all $s_1 \in \mathfrak{S}_1, s_2, s_2' \in \mathfrak{S}_2$, and $e_2 \in \mathfrak{E}_2$ such that $s_2 \xrightarrow{e_2} s_2'$ and $(s_2, s_1) \in R$ there exists some $s_1' \in \mathfrak{S}_1$ such that $s_1 \xrightarrow{\pi(e_2)} s_1'$.*

Sprenger et. al [17, Theorem 2.1] states the following theorem about the relation of traces from event systems that refine each other.

Theorem 2.52 $(\Xi_2, \mathfrak{I}_2) \sqsubseteq_\pi (\Xi_1, \mathfrak{I}_1)$ *implies* $\pi(\mathsf{traces}(\Xi_2, \mathfrak{I}_2)) \subseteq \mathsf{traces}(\Xi_1, \mathfrak{I}_1)$

Proof Let $\Xi_1 = (\mathfrak{S}_1, \mathfrak{E}_1, \rightarrow_1)$ and $\Xi_2 = (\mathfrak{S}_2, \mathfrak{E}_2, \rightarrow_2)$. Let $\mathfrak{I}_1 \subseteq \mathfrak{S}_1$ and $\mathfrak{I}_2 \subseteq \mathfrak{S}_2$.

We can prove Theorem 2.52 by structural induction over event system traces. We will show that for every trace $\sigma \in \mathsf{traces}(\Xi_2, \mathfrak{I}_2)$ its projection $\pi(\sigma)$, is in $\mathsf{traces}(\Xi_1, \mathfrak{I}_1)$.

- **Base Case:** $\sigma = \varepsilon \in \mathsf{traces}(\Xi_2, \mathfrak{I}_2)$. The empty trace is part of any event system, and therefore $\pi(\sigma) = \pi(\varepsilon) = \varepsilon \in \mathsf{traces}(\Xi_1, \mathfrak{I}_1)$. And further, for any inital state $s_2 \in \mathfrak{I}_2$ there exists there exists an initial state $s_1 \in \mathfrak{I}_1$ such that $(s_2, s_1) \in R$ for which $s_2 \xrightarrow{\varepsilon}_2 s_2$ and. $s_1 \xrightarrow{\pi(\varepsilon)}_1 s_1$.
- **Induction Hypothesis** For any trace $\sigma = \sigma' \cdot e \in \mathsf{traces}(\Xi_2, \mathfrak{I}_2)$ with $|\sigma| > |\sigma'|$, $\pi(\sigma') \in \mathsf{traces}(\Xi_1, \mathfrak{I}_1)$. And there exists an initial state $s_2 \in \mathfrak{I}_2$, a state $s_2' \in \mathfrak{S}_2$, an initial state $s_1 \in \mathfrak{I}_1$, and a state $s_1' \in \mathfrak{S}_1$ such that $(s_2, s_1) \in R$, $(s_1', s_2') \in R$, $s_2 \xrightarrow{\sigma'}_2 s_2'$, and $s_1 \xrightarrow{\pi(\sigma')}_1 s_1'$.
- **Inductive Step:** $\sigma \in \mathsf{traces}(\Xi_2, \mathfrak{I}_2)$. Since we covered the empty trace in the base case, we have $|\sigma| \geq 1$. Thus we can assume that $\sigma = \sigma' \cdot e$ for some $e \in \mathfrak{E}_2$. By the Induction Hypothesis we have that $\pi(\sigma') \in \mathsf{traces}(\Xi_1, \mathfrak{I}_1)$.

  By the induction hypothesis we have that $\pi(\sigma')$ is a trace of the event system $(\Xi_1, \mathfrak{I}_1)$. Also by induction hypthesis we have that there exist $s_2 \in \mathfrak{I}_2$, $s_2' \in \mathfrak{S}_2$, $s_1 \in \mathfrak{I}_1$, and $s_1' \in \mathfrak{S}_1$ such that $(s_2, s_1) \in R$, $(s_1', s_2') \in R$, $s_2 \xrightarrow{\sigma'}_2 s_2'$, and $s_1 \xrightarrow{\pi(\sigma')}_1 s_1'$. By definition of σ as $\sigma' \cdot e$ and the assumption that $\sigma \in \mathsf{traces}(\Xi_2, \mathfrak{I}_2)$ we have that $s_2' \xrightarrow{e}_2 s_2''$ for some $s_2'' \in \mathfrak{S}_2$. By point 2 of Definition 2.51 we have that there exists a state $s_1'' \in \mathfrak{S}_1$ such that $s_1' \xrightarrow{\pi(e)}_1 s_1''$. Thus $s_1 \xrightarrow{\pi(\sigma')}_1 s_1' \xrightarrow{e}_1 s_2''$ and also $s_1 \xrightarrow{\pi(\sigma')}_1 s_1''$. □

The next definition defines properties over event systems according to Sprenger et. al [17, Section 2.1.1]. It is analogous to the definition of properties over MFOTL traces.

Definition 2.53 *A* trace property *P is a set of traces over events* $\mathbb{E}$.

$\Xi \models P$ denotes that the event system $\mathsf{traces}(\Xi, \mathfrak{I}) \subseteq P$.

Next we recall Sprenger et. al [17, Lemma 2.2].

Lemma 2.54 *Suppose* $\Xi_1, \mathfrak{I}_1 \models P_1$ *and* $\pi(\mathsf{traces}(\Xi_2, \mathfrak{I}_2)) \subseteq \mathsf{traces}(\Xi_1, \mathfrak{I}_1)$*, then* $\Xi_2, \mathfrak{I}_2 \models \pi^{-1}(P_1)$

Proof $\Xi_1, \mathfrak{I}_1 \models P_1$ is equivalent to $\mathsf{traces}(\Xi_1, \mathfrak{I}_1) \subseteq P_1$. Thus

$$\begin{aligned}
&\pi(\mathsf{traces}(\Xi_2, \mathfrak{I}_2)) \subseteq \mathsf{traces}(\Xi_1, \mathfrak{I}_1) \subseteq P_1 \\
\Rightarrow\ &\pi^{-1}(\pi(\mathsf{traces}(\Xi_2, \mathfrak{I}_2))) \subseteq \pi^{-1}(\mathsf{traces}(\Xi_1, \mathfrak{I}_1)) \subseteq \pi^{-1}(P_1) \\
\Rightarrow\ &\mathsf{traces}(\Xi_2, \mathfrak{I}_2) \subseteq \pi^{-1}(\mathsf{traces}(\Xi_1, \mathfrak{I}_1)) \subseteq \pi^{-1}(P_1) \\
\Rightarrow\ &\mathsf{traces}(\Xi_2, \mathfrak{I}_2) \subseteq \pi^{-1}(P_1).
\end{aligned}$$

□

2.6 Legal Background

A Lex specification represents a formalization of a legal text. Any formalization always includes some level of interpretation. Writing a Lex specification requires to make decisions on which legal concepts are represented as an event, how general should events be, which aspects of the law fit into which kind of CRE rule, etc. Only by carefully auditing and reviewing a Lex specification is there a high certainty that a given specification correctly captures the essence of the law. Because the task of writing such a formalization is inherently difficult, the ability to reuse an existing specification and adapt it with minimal changes to different use cases is highly desirable. This thesis provides one method of achieving this reusability by ways of refinement.

The refinement process arguably includes even more interpretation than the initial formalization. With knowledge about a given system, some aspects of an initial specification might become redundant and others might need to be clarified based on how a system implements certain things. For example, in our ePrivacy Directive example on unsolicited communications, if we know a system never makes calls or uses a fax, rules on limiting the use of calls and fax become unnecessary.

Chapter 3

Refinement of CRE Enforcement

3.1 Introduction

In this chapter we will introduce a different definition of traces (system traces) and show that, under some conditions, it is isomorph to the (MFOTL) trace definition introduced in chapter 2. We will also define what *Refinement* for MFOTL formulae as well as Lex specifications. We will show that with our definitions, if a Lex specification A is a refinement of a Lex specification B, then A compiled to MFOTL is an MFOTL refinement of B compiled to MFOTL. We will further introduce a model of runtime enforcement as an event system. This new model will be based on the new trace definition. We will give definitions for soundness and transparency of runtime enforcement event systems. We will show that these definitions are equivalent to those of chapter 2 under the aforementioned isomorphism between system and MFOTL traces. We will adapt the definition of refinement from Sprenger et al. [17] slightly. Specifically, our mediator function will not only rely on an event being reported at a certain point in time, but also the trace (history) of previous events.

We will then go on to show that, whenever there exists a sound enforcement event system for some MFOTL formula φ, and another formula ψ refines φ, then there exists a sound enforcement event system for ψ. Separately we will show that there exists a sound enforcement event system for some MFOTL formula φ if and only if there exists a sound enforcer, as defined in chapter 2 for φ. Hublet et al. [12] have shown that an causable Lex specification compiles to an causable MFOTL formula. And Hublet et al. [9] have shown that if an MFOTL formula is causable, there exists a sound enforcer for it. Putting everything together we will have shown that if a Lex specification A refines an enforceable Lex specification B, then B is enforceable.

3.2 Alternative Enforcement Model

3.2.1 System Traces

We introduce a different trace definition as in Definition 2.6 and we will show that this new definition is at least as expressive as the old one. This definition will help us to look at runtime enforcement as an event system in Subsection 3.2.2 that works with traces in this new model. By showing that the two trace definitions are isomorphic, we will be able to show that the two runtime enforcement models are also equivalent. This will tie together the refinement of Lex specifications and in turn of CRE[MFOTL] with the refinement definition of Sprenger et al. [17].

Definition 3.1 *A* System Sequence $\sigma_{\mathbf{s}} \in \mathbb{S}_{\mathbf{s}}$ *is a finite or infinite sequence of events*

$$\langle e_i \rangle_{1 \leq i \leq k}, k \in \mathbb{N} \cup \{\infty\}$$

where for every $i \in [1, k]$,

$$e_i \in \mathsf{DB} \cup \{\mathsf{tick}\}\,.$$

The special event tick represents a clock tick in the system. As a more compact notation we use

$$e^k := \underbrace{e \cdot \ldots \cdot e}_{k \text{ times}}$$

for all $k \in \mathbb{N}$ and $e \in \mathsf{DB} \cup \{tick\}$. Note that for $k = 0$, $e^k = \varepsilon_{\mathbf{s}}$.

Definition 3.2 *The function*

$$\#\mathsf{tick} : \mathbb{S}_{\mathbf{s}} \to \mathbb{N} \cup \{\infty\}$$

$$\#\mathsf{tick}(\sigma_{\mathbf{s}}) = \begin{cases} k & \text{if } \sigma_{\mathbf{s}} = \mathsf{tick}^k \text{ for } k \geq 0 \\ k + \#\mathsf{tick}(\sigma'_{\mathbf{s}}) & \text{if } \sigma_{\mathbf{s}} = \mathsf{tick}^k \cdot D \cdot \sigma'_{\mathbf{s}} \text{ for } k \geq 0 \text{ and } D \in \mathsf{DB} \end{cases}$$

gives the number of ticks in a system sequence.

The tick function is useful to determine a timestamp in a system trace, since they are only implicit and not explicit like in MFOTL traces.

Definition 3.3 *The function*

$$\mathsf{trim} : \mathbb{S}_{\mathbf{s}} \to \mathbb{S}_{\mathbf{s}}$$

$$\mathsf{trim}(\sigma_{\mathbf{s}}) := \begin{cases} \varepsilon_{\mathbf{s}} & \text{if } \sigma_{\mathbf{s}} = \mathsf{tick}^k \text{ for } k \geq 0 \\ \mathsf{tick}^k \cdot D \cdot \mathsf{trim}(\sigma'_{\mathbf{s}}) & \text{if } \sigma_{\mathbf{s}} = \mathsf{tick}^k \cdot D \cdot \sigma'_{\mathbf{s}} \text{ for } k \geq 0 \text{ and } D \in \mathsf{DB} \end{cases}$$

removes all tick *events from the end of a system sequence that are not followed by a database.*

For sets of system sequences $A \subseteq \mathbb{S}_{\mathbf{s}}$, $\mathsf{trim}(A) := \{\mathsf{trim}(\sigma) \mid \sigma \in A\}$.

Definition 3.4 *A* System Trace $\sigma_{\mathbf{s}} \in \mathbb{T}_{\mathbf{s}} \subseteq \mathbb{S}_{\mathbf{s}}$ *is a finite or infinite system sequence where the number of ticks progresses with the number of databases (if* $\#\mathsf{DB}(\sigma_{\mathbf{s}}) = \infty$*, then* $\#\mathsf{tick}(\mathsf{trim}(\sigma_{\mathbf{s}})) = \infty$*).*

The distinction between system sequences and system traces is made, such that for infinite traces, it is guaranteed that at any point in the trace, there will eventually follow a database event.

Definition 3.5 *The function*

$$\#\mathsf{DB} : \mathbb{T}_{\mathbf{s}} \to \mathbb{N} \cup \{\infty\}$$

$$\#\mathsf{DB}(\sigma_{\mathbf{s}}) = \begin{cases} 0 & \textit{if } \sigma_{\mathbf{s}} = \mathsf{tick}^k \textit{ and } k \geq 0 \\ 1 + \#\mathsf{DB}(\sigma'_{\mathbf{s}}) & \textit{if } \sigma_{\mathbf{s}} = \mathsf{tick}^k \cdot D \cdot \sigma'_{\mathbf{s}} \textit{ for } k \geq 0 \textit{ and } D \in \mathsf{DB} \end{cases}$$

gives the number of database events in a system trace.

We say a system trace $\sigma_{\mathbf{s}} \in \mathbb{T}_{\mathbf{s}}$ is finite iff $\#\mathsf{DB}(\sigma_{\mathbf{s}}) \in \mathbb{N}$ and it is infinite iff $\#\mathsf{DB}(\sigma_{\mathbf{s}}) = \infty$. We use $\mathbb{T}_{\mathbf{s}f} \subseteq (\mathsf{DB} \cup \{\mathsf{tick}\})^*$, $\mathbb{T}_{\mathbf{s}\omega} \subseteq (\mathsf{DB} \cup \{\mathsf{tick}\})^\infty$, and $\mathbb{T}_{\mathbf{s}} = \mathbb{T}_{\mathbf{s}f} \cup \mathbb{T}_{\mathbf{s}\omega}$ to denote the sets of finite system traces, infinite system traces, and all system traces respectively. $\varepsilon_{\mathbf{s}}$ denotes an empty system trace.

Lemma 3.6 *For any infinite system trace* $\sigma_{\mathbf{s}} \in \mathbb{T}_{\mathbf{s}\omega}$*,* $\sigma_{\mathbf{s}} = \mathsf{trim}(\sigma_{\mathbf{s}})$*.*

This follows directly from our definition for infinite system traces. At any point within an infinite trace, there must eventually follow at least one database, because otherwise there would not be an infinite number of databases in the trace. Lemma 3.6 can be rewritten as $\mathbb{T}_{\mathbf{s}\omega} = \mathsf{trim}(\mathbb{T}_{\mathbf{s}\omega})$.

Next we define conversion functions between trimmed system traces and MFOTL traces. And then we will show that the conversion functions together define an isomorphism between MFOTL traces and trimmed system traces. The limitation on trimmed system traces is necessary, because MFOTL traces do not capture any timepoints at which no database is logged.

Definition 3.7 *The* Finite Extended System Trace Converter Function *converts system traces starting at some timestamp to MFOTL traces.*

$$\mathsf{c}' : \mathbb{N} \times \mathbb{T}_{\mathbf{s}} \to \mathbb{T}$$

$$\mathsf{c}'(\tau, \sigma_{\mathbf{s}}) := \begin{cases} \varepsilon & \textit{if } \mathsf{trim}(\sigma_{\mathbf{s}}) = 0 \\ (\tau + k, D) \cdot \mathsf{c}'(\tau + k, \sigma'_{\mathbf{s}}) & \textit{if } \sigma_{\mathbf{s}} = \mathsf{tick}^k \cdot D \cdot \sigma'_{\mathbf{s}} \\ & \textit{with } k \geq 0, D \in \mathsf{DB} \end{cases}$$

Definition 3.8 *The* System Trace Converter Function *converts system traces starting at timestamp* 0 *to MFOTL traces.*

$$\mathsf{c} : \mathbb{T}_{\mathbf{s}} \to \mathbb{T}$$

$$\mathsf{c}(\sigma_{\mathbf{s}}) := \mathsf{c}'(0, \sigma_{\mathbf{s}}).$$

Definition 3.9 *The extended* MFOTL Trace Converter Function *converts MFOTL traces starting at some timestamp to system traces.*

$$\mathsf{d}' : \mathbb{N} \times \mathbb{T} \to \mathsf{trim}(\mathbb{T}_\mathbf{s})$$

$$\mathsf{d}'(\tau, \sigma) := \begin{cases} \varepsilon_\mathbf{s} & \textit{if } \sigma = \varepsilon \\ \mathsf{tick}^k \cdot D \cdot \mathsf{d}'(\tau + k, \sigma') & \textit{if } \sigma = (\tau + k, D) \cdot \sigma' \\ & \textit{with } k \geq 0 \end{cases}$$

Definition 3.10 *The* MFOTL Trace Converter Function *converts MFOTL traces starting at timestamp* 0 *to system traces.*

$$\mathsf{d} : \mathbb{T} \to \mathsf{trim}(\mathbb{T}_\mathbf{s})$$

$$\mathsf{d}(\sigma) := \mathsf{d}'(0, \sigma).$$

It is important to note that $\mathsf{d}'(\tau, \sigma)$ is only defined, if $\tau \leq \mathsf{fts}(\sigma)$ or $\sigma = \varepsilon$. d however, is defined for all $\sigma \in \mathbb{T}$, since timestamps in any trace must increase monotonically, and the timestamp argument passed to d' is 0. To see that the images of both d and d' for all allowed inputs is $\mathsf{trim}(\mathbb{T}_\mathbf{s})$ and not $\mathbb{T}_\mathbf{s}$, note that a tick is only added when the trace is not yet empty, thus any inserted tick will eventually be followed by some database D. For any set of MFOTL traces $A \subseteq \mathbb{T}$ and any set of system traces $B \in \mathbb{T}_\mathbf{s}$ and timestamp $\tau \in \mathbb{N}$, we write

$$\begin{aligned} \mathsf{c}(A) &:= \{\mathsf{c}(\sigma_\mathbf{s}) \mid \sigma_\mathbf{s} \in A\}, \\ \mathsf{c}'(\tau, A) &:= \{\mathsf{c}'(\tau, \sigma_\mathbf{s}) \mid \sigma_\mathbf{s} \in A\}, \\ \mathsf{d}(B) &:= \{\mathsf{d}(\sigma) \mid \sigma \in B\}, \textit{and} \\ \mathsf{d}'(\tau, B) &:= \{\mathsf{d}'(\tau, \sigma) \mid \sigma \in B\}. \end{aligned}$$

Theorem 3.11 *The trace converter functions* c *and* d *define a bijective mapping between MFOTL traces* $\mathbb{T}$ *and trimmed system traces* $\mathsf{trim}(\mathbb{T}_\mathbf{s})$ *and are each others inverses.*

$$\forall \sigma \in \mathbb{T}.\ \sigma = \mathsf{d}(\mathsf{c}(\sigma))$$

and

$$\forall \sigma_\mathbf{s} \in \mathsf{trim}(\mathbb{T}_\mathbf{s}).\ \sigma_\mathbf{s} = \mathsf{c}(\mathsf{d}(\sigma))$$

The intuition behind the necessity of trimming system traces to make them fully comparable with MFOTL traces is that MFOTL traces only capture some last timestamp at which a database was logged, while system traces can capture a current timestamp (the total number of ticks in a system trace) which may be greater than the last time stamp at which a database was logged (the number of ticks up until the last database).

Proof The proof for Theorem 3.11 proceeds by splitting it into two properties, one for each direction. We want to show the two following equations.

$$\forall \sigma \in \mathbb{T}. \qquad \sigma = \mathsf{d}(\mathsf{c}(\sigma)), \tag{3.1}$$

$$\forall \sigma_{\mathsf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathsf{s}}). \qquad \sigma_{\mathsf{s}} = \mathsf{c}(\mathsf{d}(\sigma_{\mathsf{s}})), \tag{3.2}$$

Since c and d are defined with c' and d' respectively, the proof will proceed by showing the following two, more general, properties for all $\tau \in \mathbb{N}$.

$$\forall \sigma \in \mathbb{T}.\ (\sigma = \varepsilon \lor \tau \leq \mathsf{fts}(\sigma)) \to \mathsf{c}'(\tau, \mathsf{d}'(\tau, \sigma)) = \sigma, \tag{3.3}$$

$$\forall \sigma_{\mathsf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathsf{s}}).\ \mathsf{d}'(\tau, \mathsf{c}'(\tau, \sigma_{\mathsf{s}})) = \sigma_{\mathsf{s}}, \tag{3.4}$$

When fixing $\tau = 0$ in these two equations respectively, the precondition in Equation 3.3 evaluates to true, and the two equations become equivalent to Equation 3.1 and Equation 3.4. Thus Equation 3.3 implies Equation 3.1 and Equation 3.4 implies Equation 3.2.

For each direction we will first show that they hold for finite traces and then make the argument they also hold for infinite traces.

1. **MFOTL Traces** (Equations 3.1 and 3.3)

   As stated above, we begin by showing Equation 3.3 for finite traces $\sigma \in \mathbb{T}_f$. In a second step we will then make the argument why it also holds for infinite MFOTL traces. We proof this by structural induction over traces $\sigma \in \mathbb{T}_f$ for all $\tau \in \mathbb{N}$ where either $\tau \leq \mathsf{fts}(\sigma)$ or $\sigma = \varepsilon$.

   Base Case: $\sigma = \varepsilon$ (and some $\tau \in \mathbb{N}$)

   $$\mathsf{c}'(\tau, \mathsf{d}'(\tau, \sigma)) = \mathsf{c}'(\tau, \mathsf{d}(\tau, \varepsilon)) \overset{3.9}{=} \mathsf{c}'(\tau, \varepsilon_{\mathsf{s}}) \overset{3.7}{=} \varepsilon = \sigma$$

   Induction Hypothesis: For every true suffix σ' of a trace $\sigma \in \mathbb{T}_f$ that is not the entire trace σ, i.e. $\sigma = \sigma'' \cdot \sigma'$ and $\sigma'' \neq \varepsilon$, and for every timestamp $\tau \in \mathbb{N}$ which is less than the first timestamp ($\tau \leq \mathsf{fts}(\sigma')$) in σ', unless $\sigma' = \varepsilon$ it holds that

   $$\sigma' = \mathsf{c}'(\tau, \mathsf{d}'(\tau, \sigma')).$$

   Induction Step: Let $\sigma \neq \varepsilon$ and $\tau \leq \mathsf{fts}(\sigma)$. Because σ is not empty, $\sigma = (\tau + k, D) \cdot \sigma'$ for some $k \geq 0$, some $D \in \mathsf{DB}$ and a suffix $\sigma' \in \mathbb{T}_f$.

   $$\begin{aligned} \mathsf{c}'(\tau, \mathsf{d}'(\tau, \sigma)) &= \mathsf{c}'(\tau, \mathsf{d}'(\tau, (\tau + k, D) \cdot \sigma')) \\ &\overset{3.9}{=} \mathsf{c}'(\tau, \mathsf{tick}^k \cdot D \cdot \mathsf{d}'(\tau + k, \sigma')) \\ &\overset{3.7}{=} (\tau + k, D) \cdot \mathsf{c}'(\tau + k, \mathsf{d}'(\tau + k, \sigma')) \\ &\overset{\text{IH}}{=} (\tau + k, D) \cdot \sigma' = \sigma \end{aligned}$$

Finally we show how this extends to infinite traces. Let $\sigma \in \mathbb{T}_\omega$ be an infinite MFOTL trace. And let $\sigma_{|\ldots i}$ be the finite prefix with the first i time points of σ. With the finite case, we have shown that for any $i \in \mathbb{N}$, $\sigma \in \mathbb{T}_\omega$, and $\tau \leq \mathsf{fts}(\sigma_{|\ldots i})$ (in the infinite case, a first timestamp always exists) $\sigma_{..i} = \mathsf{c}'(\tau, \mathsf{d}'(\tau, \sigma_{|\ldots i}))$. Then in the limit

$$\sigma = \lim_{i \to \infty} \sigma_{|\ldots i} = \lim_{i \to \infty} \mathsf{c}'(\tau, \mathsf{d}'(\tau, \sigma_{|\ldots i})) = \mathsf{c}'(\tau, \mathsf{d}'(\tau, \sigma)).$$

We have now shown 3.1 for both finite and infinite traces, which directly implies 3.3.

2. **System Traces** (Equations 3.2 and 3.4) For this direction we first prove that Equation 3.4 holds for finite system traces $\sigma_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}f})$. And in a second step will show that it also holds for infinite system traces $\sigma_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}\omega})$. We recall that whether or not a system trace is finite or infinite depends on the number of database events in the trace.

   This first part of the proof proceeds by structural induction over finite and trimmed system traces $\sigma_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}f})$ for all $\tau \in \mathbb{N}$.

   a) **Base Case** $\sigma_{\mathbf{s}} = \varepsilon_{\mathbf{s}}$

   $$\mathsf{d}'(\tau, \mathsf{c}'(\tau, \sigma_{\mathbf{s}})) = \mathsf{d}'(\tau, \mathsf{c}'(\tau, \varepsilon_{\mathbf{s}})) \stackrel{3.7}{=} \mathsf{d}'(\tau, \varepsilon) \stackrel{3.9}{=} \varepsilon_{\mathbf{s}} = \sigma_{\mathbf{s}}$$

   b) **Induction Hypothesis** For every true suffix $\sigma'_{\mathbf{s}}$ of a trimmed system trace $\sigma_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}f})$ that is not the entire trace $\sigma_{\mathbf{s}}$, i.e. $\sigma_{\mathbf{s}} = \sigma''_{\mathbf{s}} \cdot \sigma'_{\mathbf{s}}$ and $\sigma''_{\mathbf{s}} \neq \varepsilon$, and for every timestamp $\tau \in \mathbb{N}$ it holds that

   $$\sigma'_{\mathbf{s}} = \mathsf{d}'(\tau, \mathsf{c}'(\tau, \sigma'_{\mathbf{s}})).$$

   c) **Induction Step**

   Let $\sigma_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}f})$, with $\sigma_{\mathbf{s}} \neq \varepsilon_{\mathbf{s}}$. Then $\sigma_{\mathbf{s}} = \mathsf{tick}^k \cdot D \cdot \sigma'_{\mathbf{s}}$ for some $k \geq 0$, $D \in \mathsf{DB}$, and a suffix $\sigma'_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}f})$.

   $$\begin{aligned} \mathsf{d}'(\tau, \mathsf{c}'(\tau, \sigma_{\mathbf{s}})) &= \mathsf{d}'(\tau, \mathsf{c}'(\tau, \mathsf{tick}^k \cdot D \cdot \sigma'_{\mathbf{s}})) \\ &\stackrel{3.7}{=} \mathsf{d}'(\tau, (\tau + k, D) \cdot \mathsf{c}'(\tau + k, \sigma'_{\mathbf{s}})) \\ &\stackrel{3.9}{=} \mathsf{tick}^k \cdot D \cdot \mathsf{d}'(\tau + k, \mathsf{c}'(\tau + k, \sigma'_{\mathbf{s}})) \\ &\stackrel{IH}{=} \mathsf{tick}^k \cdot D \cdot \sigma'_{\mathbf{s}} = \sigma_{\mathbf{s}} \end{aligned}$$

   Finally we show how this extends to infinite system traces. Let $\sigma_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}\omega})$ be an infinite system trace. And let $\sigma_{\mathbf{s}|\ldots i}$ be the finite prefix of $\sigma_{\mathbf{s}}$ ending with the i-th database event. By this definition

$\sigma_{\mathbf{s}|\ldots i}$ is in $\mathsf{trim}(\mathbb{T}_{\mathbf{s}f})$. With the finite case, we have shown that for any $i \in \mathbb{N}$, $\sigma_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}\omega})$, and $\tau \in \mathbb{N}$ $\sigma_{\mathbf{s}|\ldots i} = \mathsf{d}'(\tau, \mathsf{c}'(\tau, \sigma_{\mathbf{s}|\ldots i}))$. Then in the limit

$$\sigma_{\mathbf{s}} = \lim_{i\to\infty} \sigma_{\mathbf{s}|\ldots i} = \lim_{i\to\infty} \mathsf{d}'(\tau, \mathsf{c}'(\tau, \sigma_{\mathbf{s}|\ldots i})) = \mathsf{d}'(\tau, \mathsf{c}'(\tau, \sigma_{\mathbf{s}})). \qquad \square$$

Intuitively c and c' count the clock ticks and store them explicitly as integer timestamps. For both trace definitions, a property is a subset of all possible traces. For conversion of properties we use c with arguments being sets of traces in which case c simply gets mapped on each element of the set. As in Sprenger et. al [17] and Abadi and Lamport [3] traces of this second form can be extended with a stuttering step skip, where traces are equivalent if they are the same with all skip steps removed. In particular this means that sets of system traces (such as trace properties) are closed under the addition and removal of skip events. Thus the set of events in system traces $\mathbb{T}_{\mathbf{s}}$ gets extended to be

$$\mathsf{DB} \cup \{\mathsf{tick}, \mathsf{skip}\}\,.$$

Since skip events do not meaningfully modify a system trace, the isomorphism from Theorem 3.11 works if we remove all skip events before doing the trace conversion.

With the isomorphism between MFOTL traces as defined in prior work and Section 2 and trimmed system traces we can define MFOTL semantics for infinite system traces as follows.

Example 3.12 *If we recall the MFOTL trace σ from Example 2.10, we can now convert it into a system trace $\sigma_{\mathbf{s}} := \mathsf{c}(\sigma)$.*

$$\begin{aligned}\sigma_{\mathbf{s}} = \langle &\{\mathsf{giveEmailConsent}(\text{"}Alice\text{"}, \text{"}Account\ Updates\text{"})\}, \mathsf{tick},\\ &\{\mathsf{giveEmailConsent}(\text{"}Bob\text{"}, \text{"}Marketing\text{"})\}, \mathsf{tick},\\ &\{\mathsf{revokeEmailConsent}(\text{"}Alice\text{"}, \text{"}Account\ Updates\text{"}),\\ &\quad \mathsf{giveEmailConsent}(\text{"}Bob\text{"}, \text{"}Account\ Updates\text{"})\},\\ &\mathsf{tick}, \mathsf{tick}, \mathsf{tick}, \mathsf{tick}, \mathsf{tick}, \mathsf{tick}, \mathsf{tick}, \mathsf{tick}, \emptyset\\ &\rangle\end{aligned}$$

Definition 3.13 *The semantics of MFOTL for a fixed, infinite system trace $\sigma_{\mathbf{s}} \in \mathbb{T}_{\mathbf{s}\omega}$, a valuation v, time-point i, and MFOTL formula φ is defined as*

$$v, i \vDash_{\sigma_{\mathbf{s}}} \varphi \quad \text{iff} \quad v, i \vDash_{\mathsf{c}(\sigma_{\mathbf{s}})} \varphi.$$

3.2.2 Runtime Enforcement as an Event System

In this subsection we define an alternative runtime enforcement model than that presented in Section 2.2.1. and by Hublet et al. [9]. We want to describe runtime enforcement as an event system, such that we can show that our definition of CRE[MFOTL] refinement corresponds to the notion of refinement for event systems as described by Sprenger et al. [17]. Describing runtime enforcement as an event system gives a model comparable to the edit automata approach by Ligatti et al. [14], since both edit automata and event systems are forms of state machines.

The event system model of runtime enforcement works with system traces as introduced in Section 3.2.1. At the end of this seciont, with the isomorphism between MFOTL traces and trimmed system traces from Theorem 3.11, we will show that the two runtime enforcement models are equivalent.

Our model of proactive enforcement as an event system relies on two update functions, one for the proactive and one for the reactive case

Definition 3.14 *A* Reactive Update Function

$$\delta : \mathbb{T}_{\mathbf{s}f} \times \mathsf{DB} \to \mathsf{DB}$$

modifies a reported set of events by suppressing (deleting) and causing (inserting) certain events if needed. Where for all $\sigma \in \mathbb{T}_{\mathbf{s}f}$ and $D \in \mathsf{DB}$

- $D \setminus \delta(\sigma, D) \in \mathsf{DB}_{\mathbb{S}}$
- $\delta(\sigma, D) \setminus D \in \mathsf{DB}_{\mathbb{C}}$

Definition 3.15 *A* Proactive Update Function

$$\gamma : \mathbb{T}_{\mathbf{s}f} \to \mathsf{DB}_{\mathbb{C}} \cup \{\mathsf{skip}\}$$

indicates whether or not some events must be caused (inserted) before a clock tick happens.

Definition 3.16 *The* Enforcement State Space $\mathfrak{S} = \mathbb{T}_{\mathbf{s}}$ *consists of all system traces (that may include* skip*).*

Definition 3.17 *The single initial state is the empty trace $\varepsilon_{\mathbf{s}}$.*

$$\mathfrak{I} = \{\varepsilon_{\mathbf{s}}\}$$

Definition 3.18 *The* Event Space

$$\mathfrak{E} = \mathsf{DB} \cup \{\mathsf{tick}, \mathsf{skip}\}$$

consists of sets of events or the two special events tick *and* skip*.*

Definition 3.19 *The* Transition Relation *is a subset*

$$\to \subseteq \mathbb{T}_{\mathbf{s}} \times (\mathsf{DB} \cup \{\mathsf{tick}, \mathsf{skip}\}) \times \mathbb{T}_{\mathbf{s}}$$

and it is defined by the following two transition rules.

$$\sigma \xrightarrow{\delta(\sigma,D)} \sigma \cdot \delta(\sigma, D) \qquad \textit{(Step-R)}$$

$$\sigma \xrightarrow{\gamma(\sigma)\cdot\mathsf{tick}} \sigma \cdot \gamma(\sigma) \cdot \mathsf{tick} \qquad \textit{(Step-P)}$$

We write $\sigma \xrightarrow{e} \sigma'$ as a shorthand for $(\sigma, e, \sigma') \in\rightarrow$. Further for events $e_1, e_2 \in \mathfrak{E}$, $\sigma \xrightarrow{e_1 \cdot e_2} \sigma'$ iff there exists σ'' such that $\sigma \xrightarrow{e_1} \sigma''$ and $\sigma'' \xrightarrow{e_2} \sigma'$. This syntax can be extend to sequences of events (i.e. system traces) $\rho_1, \rho_2 \in \mathbb{T}_{\mathbf{s}}$, $\sigma \xrightarrow{\rho_1 \cdot \rho_2} \sigma'$ iff there exists σ'' such that $\sigma \xrightarrow{\rho_1} \sigma''$ and $\sigma'' \xrightarrow{\rho_2} \sigma'$.

Definition 3.20 *An* Enforcement Event System $\Xi = (\mathfrak{S}, \mathfrak{E}, \delta, \gamma)$ *is a tuple consisting of a state space* $\mathfrak{S} := \mathbb{T}_{\mathbf{s}}$, *event space* $\mathfrak{E} = \mathsf{DB} \cup \{\mathsf{tick}, \mathsf{skip}\}$, *a reactive update function* $\delta : \mathbb{T}_{\mathbf{s}f} \times \mathsf{DB} \rightarrow \mathsf{DB}$ *and a proactive update function* $\gamma : \mathbb{T}_{\mathbf{s}f} \rightarrow \mathsf{DB}_{\mathsf{C}} \cup \{\mathsf{skip}\}$. *Where the two update functions define a transition relation as defined in Definition 3.19.*

We use the same definition of event traces for an enforcement event system Ξ with initial states $\mathfrak{I}$ as for event systems inin Sprenger et al. [17] (Section 2.4), states $\mathfrak{I}$. Our described system has the property that if the only initial state is the empty trace $\varepsilon_{\mathbf{s}}$, the final state of the event system and the event trace of the system are identical.

We define soundness notions for enforcement event systems that are analogous to soundness of an enforcer in Subsection 2.2.1. Sprenger et al. [17] define traces as always being finite sequences of events. We extend that definition to include both finite and infinite event sequences. The function $\mathsf{traces}(\Xi, \mathfrak{I})$ then denotes both all finite and infinite event sequences of the event system Ξ starting in any initial state in $\mathfrak{I}$. $\mathsf{traces}_f(\Xi, \mathfrak{I})$ denotes only the finite traces and $\mathsf{traces}_\omega(\Xi, \mathfrak{I})$ only the infinite traces. We also extend the definition of trace properties to include both finite and infinite sequences. We use the following notations if we specifically talk about trace properties that are strictly finite or infinite respectively: $\Xi, \mathfrak{I} \vDash_f P$ iff $\mathsf{traces}_f(\Xi, \mathfrak{I}) \subseteq P$ and $\Xi, \mathfrak{I} \vDash_\omega P$ iff $\mathsf{traces}_\omega(\Xi, \mathfrak{I}) \subseteq P$. Like Sprenger et. al [17] we will omit the set braces for initial states, if the set is a singleton. Note that system traces are only considered infinite if they are infinite with all skips removed and trimmed (ticks not followed by a database removed).

Next we will show that any enforcer as in Hublet et al. [9] has a corresponding enforcement event system. To simplify this we define a helper function, that can compute the state of an enforcer after having enforced some trace as well as potentially some "leftover" ticks.

Definition 3.21 *The* Enforcer State Function

$$\mathsf{state} : \mathcal{S} \times \mathbb{T}_{\mathbf{s}} \rightarrow \mathcal{S}$$

gives the state starting in some initial state, after having enforced the corresponding MFOTL trace of a system trace as well as the ticks that may or may not be "dangling" at the end of the system trace. This function is defined by essentially simulating the run *function of Algorithm 1. The simulation uses a helper function*

$$\mathsf{state}' : \mathcal{S} \times \mathbb{T}_{\mathbf{s}} \times \mathbb{T}_{\mathbf{s}} \to \mathcal{S}$$

$$\mathsf{state}'(s, \sigma_{\mathbf{s}1}, \sigma_{\mathbf{s}2}) :=$$

$$\begin{cases} s & \textit{if } \sigma_{\mathbf{s}2} = \varepsilon_{\mathbf{s}} \\ \mathsf{state}'(s', \sigma_{\mathbf{s}1} \cdot D_{\mathbb{C}} \cdot \mathsf{tick}, \sigma'_{\mathbf{s}2}) & \textit{if } \sigma_{\mathbf{s}2} = \mathsf{tick} \cdot \sigma'_{\mathbf{s}2} \\ & \textit{and } (\mathsf{PCom}(D_{\mathbb{C}}), s') \\ & \quad = \mu(\mathsf{c}(\sigma_{\mathbf{s}1}), s, \#\mathsf{tick}(\sigma_{\mathbf{s}1}), \mathsf{tick}) \\ \mathsf{state}'(s', \sigma_{\mathbf{s}1} \cdot \mathsf{tick}, \sigma'_{\mathbf{s}2}) & \textit{if } \sigma_{\mathbf{s}2} = \mathsf{tick} \cdot \sigma'_{\mathbf{s}2} \\ & \textit{and } (\mathsf{NoCom}, s') \\ & \quad = \mu(\mathsf{c}(\sigma_{\mathbf{s}1}), s, \#\mathsf{tick}(\sigma_{\mathbf{s}1}), \mathsf{tick}) \\ \mathsf{state}'(s', \sigma_{\mathbf{s}1} \cdot (D \setminus D_{\mathbb{S}} \cup D_{\mathbb{C}}), \sigma'_{\mathbf{s}2}) & \textit{if } \sigma_{\mathbf{s}2} = D \cdot \sigma'_{\mathbf{s}2} \\ & \textit{and } (\mathsf{RCom}(D_{\mathbb{C}}, D_{\mathbb{S}}), s') \\ & \quad = \mu(\mathsf{c}(\sigma_{\mathbf{s}1}), s, \#\mathsf{tick}(\sigma_{\mathbf{s}1}), D) \end{cases}$$

Then the state *function is defined as follows for any enforcer state $s \in \mathcal{S}$ and system trace $\sigma_{\mathbf{s}} \in \mathbb{T}_{\mathbf{s}}$*

$$\mathsf{state}(s, \sigma_{\mathbf{s}}) := \mathsf{state}'(s, \varepsilon_{\mathbf{s}}, \sigma_{\mathbf{s}}).$$

Definition 3.22 *The* induced enforcement event system *of an enforcer $\mathcal{E} = (\mathcal{S}, s_0, \mu)$ is defined as $\Xi(\mathcal{E}) := (\mathbb{T}_{\mathbf{s}}, \mathsf{DB} \cup \{\mathsf{tick}, \mathsf{skip}\}, \delta, \gamma)$, with δ and γ defined as follows*

$$\delta(\sigma_{\mathbf{s}}, D) := D \setminus D_{\mathbb{S}} \cup D_{\mathbb{C}}$$
$$\qquad \textit{with } \mu(\mathsf{c}(\sigma_{\mathbf{s}}), \mathsf{state}(s_0, \sigma_{\mathbf{s}}), \#\mathsf{tick}(\sigma_{\mathbf{s}}), D) = (\mathsf{RCom}(D_{\mathbb{C}}, D_{\mathbb{S}}), s)$$

$$\gamma(\sigma_{\mathbf{s}}) := \begin{cases} \mathsf{skip} & \textit{if } \mu(\mathsf{c}(\sigma_{\mathbf{s}}), \mathsf{state}(s_0, \sigma_{\mathbf{s}}), \#\mathsf{tick}(\sigma_{\mathbf{s}}), D) = (\mathsf{NoCom}, s) \\ D_{\mathbb{C}} & \textit{if } \mu(\mathsf{c}(\sigma_{\mathbf{s}}), \mathsf{state}(s_0, \sigma_{\mathbf{s}}), \#\mathsf{tick}(\sigma_{\mathbf{s}}), D) = (\mathsf{PCom}(D_{\mathbb{C}}), s) \end{cases}$$

Definition 3.23 *An enforcement event system $\Xi = (\mathbb{T}_{\mathbf{s}}, \mathsf{DB} \cup \{\mathsf{tick}, \mathsf{skip}\}, \delta, \gamma)$ is* sound *with respect to some property $P \subseteq \mathbb{T}_{\mathbf{s}\omega}$ iff*

$$\Xi, \varepsilon_{\mathbf{s}} \models P.$$

Equivalently an enforcement event system Ξ and the initial state $\varepsilon_{\mathbf{s}}$ are sound for a property $P \subseteq \mathbb{T}_{\mathbf{s}\omega}$ iff $\mathsf{traces}_{\omega}(\Xi, \varepsilon_{\mathbf{s}}) \subseteq P$. This soundness definition difers from Sprenger et al. [17] in that it is over infinite traces, where Sprenger et al. define soundness over finite traces. We however want to use infinite traces, because the enforcement model of Hublet et al. [9] defines soundenss with infinite traces. This deviation from Sprenger et al. [17] is fine to make, because any set of infinite traces could be converted into an (infinitely large) set of all finite prefixes of the inifinite traces.

Definition 3.24 *An enforcement event system Ξ is sound for a formula φ iff it is sound for the property $\mathcal{L}(\varphi)$.*

Theorem 3.25 *If an enforcer $\mathcal{E} = (\mathcal{S}, s_0, \mu)$ is sound for a property $P \subseteq \mathbb{T}_\omega$, then the induced enforcement event system $\Xi(\mathcal{E})$ is sound for the property $\mathsf{c}(P) \subseteq \mathbb{T}_{\mathbf{s}\omega}$.*

Before we can prove Theorem 3.25, we need the following two lemmata. The proofs for which can be found in Appendix 6 and 6.

Lemma 3.26 *For all $\sigma_\mathbf{s}$, $\tau \in \mathbb{N}$ and $D \in \mathsf{DB}$,*

$$\mathsf{c}'(\tau, \sigma_\mathbf{s} \cdot D) = \mathsf{c}'(\tau, \sigma_\mathbf{s}) \cdot \mathsf{c}(\tau + \#\mathsf{tick}(\sigma_\mathbf{s}), D)$$

Lemma 3.27 *Let $\mathcal{E} = (\mathcal{S}, s_0, \mu)$ be an enforcer. For any system trace $\sigma_\mathbf{s} \in \mathsf{traces}(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})$ there exists a trace $\sigma \in \mathbb{T}$ such that $\mathsf{c}(\sigma_\mathbf{s}) = \mathcal{E}(\sigma)$.*

Proof We will show that for all $\sigma_\mathbf{s} \in \mathsf{traces}_f(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})$ there exists a trace $\sigma'_\mathbf{s} \in \mathsf{c}(P)$ of which $\sigma_\mathbf{s}$ is a prefix (i.e. $\sigma_\mathbf{s} = \sigma'_{\mathbf{s}|\ldots \leq |\sigma_\mathbf{s}|}$).

Since by definition $\mathsf{traces}_f(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})$ contains all finite prefixes of $\mathsf{traces}_\omega(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})$, if $\mathsf{traces}_\omega(\Xi(\mathcal{E}), \varepsilon_\mathbf{s}) \not\subseteq \mathsf{c}(P)$, there would exist some contains all finite prefixes of $\sigma_\mathbf{s} \in \mathsf{traces}_f(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})$ which is not a prefix for any trace $\sigma'_\mathbf{s} \in \mathsf{c}(P)$. Thus if we show that every finite prefix $\sigma_\mathbf{s} \in \mathsf{traces}_f(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})$ is also a finite prefix for at least one trace $\sigma'_\mathbf{s} \in \mathsf{c}(P)$, we will have shown that $\sigma_\mathbf{s} \in \mathsf{traces}_\omega(\Xi(\mathcal{E}), \varepsilon_\mathbf{s}) \subseteq \sigma'_\mathbf{s} \in \mathsf{c}(P)$.

This property is a consequence of Lemma 3.27. In Lemma 3.27 we have shown that any enforcement event system trace corresponds to a trace produced by the enforcer. All those traces "produced" by the enforcer fall into P, by the fact that the enforcer is sound for P. and since we have shown that for all system traces $\sigma_\mathbf{s} \in \mathsf{traces}_f(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})$, there exists a trace $\sigma \in \mathbb{T}$ such that $\mathsf{c}(\sigma_\mathbf{s}) = \mathcal{E}(\sigma)$, we have also shown that $\mathsf{c}(\sigma_\mathbf{s}) \in P$ or equivalently $\mathsf{c}(\mathsf{traces}(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})) \subseteq P$. This does also imply $\mathsf{traces}(\Xi(\mathcal{E}), \varepsilon_\mathbf{s}) \subseteq \mathsf{c}(P)$. □

3.3 Refinement

3.3.1 Enforcement Event System Refinement

For Enforcement Event Systems we will now define refinement via a mediator function and a simulation relation as in Sprenger et. al [17].

Definition 3.28 *An enforcement event system*

$$\Xi_C = (\mathbb{T}_{\mathbf{s}C}, \mathsf{DB}_C \cup \{\mathsf{tick}, \mathsf{skip}\}, \delta_C, \gamma_C)$$

for some signature $\Sigma_C = (\mathbb{D}, \mathbb{E}_C, \mathbb{F}, \iota_C)$ with initial states $\mathfrak{I}_C \subseteq \mathbb{T}_C$ refines an enforcement event system $\Xi_A = (\mathbb{T}_{\mathbf{s}A}, \mathsf{DB}_A \cup \{\mathsf{tick}, \mathsf{skip}\}, \delta_A, \gamma_A)$ for some signature $\Sigma_A = (\mathbb{D}, \mathbb{E}_A, \mathbb{F}, \iota_A)$ with initial states $\mathfrak{I}_A \subseteq \mathfrak{S}_A$ via a mediator function

$$\pi : \mathbb{T}_C \times (\mathsf{DB}_C \cup \{\mathsf{tick}, \mathsf{skip}\}) \to (\mathsf{DB}_A \cup \{\mathsf{tick}, \mathsf{skip}\})$$

iff there is a simulation relation

$$R \subseteq \mathbb{T}_C \times \mathbb{T}_A$$

such that

1. *for all $\sigma_C \in \mathfrak{I}_C$ there exists some $\sigma_A \in \mathfrak{I}_A$ such that $(\sigma_C, \sigma_A) \in R$, and*
2. *for all $\sigma_A \in \mathfrak{S}_A$, $\sigma_C, \sigma'_C \in \mathfrak{S}_C$, and $e_C \in \mathfrak{E}_C$ such that $(\sigma_C, \sigma_A) \in R$ and $\sigma_C \xrightarrow{e_C} \sigma'_C$ there exists σ'_A such that $(\sigma'_C, \sigma'_A) \in R$ and $\sigma_A \xrightarrow{\pi(\sigma_C, e_C)} \sigma'_A$.*

It is important to highlight, that our definition of a mediator function differs from that in Sprenger et al. [17] in that we include a second argument for the trace up to the current point in time. The reason for this is how we define MFOTL and Lex refinement in the following sections. Without going into the details, on a high level MFOTL refinements can define let bindings for abstract events, thus ”concretizing” them. And let bindings allow for arbitrary MFOTL formulae on their right hand side which can of course depend on the past. That is why we need the history of events as an additional argument for our mediator function.

Definition 3.29 *A mediator function*

$$\pi : \mathbb{T}_C \times (\mathsf{DB}_C \cup \{\mathsf{tick}, \mathsf{skip}\}) \to (\mathsf{DB}_A \cup \{\mathsf{tick}, \mathsf{skip}\})$$

can be extended to a trace transducer function

$$\vec{\pi} : \mathbb{T}_C \to \mathbb{T}_A$$

as follows

$$\vec{\pi}(\sigma_{\mathbf{s}}) := \begin{cases} \varepsilon_{\mathbf{s}} & \text{if } \sigma_{\mathbf{s}} = \varepsilon_{\mathbf{s}} \\ \vec{\pi}(\sigma'_{\mathbf{s}}) \cdot \pi(\vec{\pi}(\sigma'_{\mathbf{s}}), e) & \text{if } \sigma_{\mathbf{s}} = \sigma'_{\mathbf{s}} \cdot e \\ & \text{for some } e \in \mathsf{DB}_C \cup \{\mathsf{tick}, \mathsf{skip}\} \end{cases}$$

The following theorem is analogous to Theorem 2.52 [17, Theorem 2.1].

Theorem 3.30 $(\Xi_C, \mathfrak{I}_C) \sqsubseteq_\pi (\Xi_A, \mathfrak{I}_A)$ *implies* $\vec{\pi}(\mathsf{traces}(\Xi_C, \mathfrak{I}_C)) \subseteq \mathsf{traces}(\Xi_A, \mathfrak{I}_A)$.

The proof for Theorem 3.30 is analogous to the proof for Theorem 2.52.

The following lemma is analogous to Lemma 2.54 [17, Lemma 2.2].

Lemma 3.31 *Suppose* $\Xi_A, \mathfrak{I}_A \vDash P$ *and* $\vec{\pi}(\mathsf{traces}(\Xi_C, \mathfrak{I}_C)) \subseteq \mathsf{traces}(\Xi_A, \mathfrak{I}_A)$ *then* $\Xi_C, \mathfrak{I}_C \vDash \vec{\pi}^{-1}(P)$

The proof for Lemma 3.31 is analogous to the proof for Lemma 2.54.

$$\frac{\mathsf{let}_d \notin \mathsf{dom}\,\Gamma}{\Gamma \vdash e(\bar{x}) : \mathsf{mon}(e)^{+}}\,\mathbb{E}^{+}_{\mathsf{mon}} \qquad \frac{\mathsf{let}_d \in \mathsf{dom}\,\Gamma \quad \Gamma(\mathsf{let}_d) = \varphi \quad \Gamma \setminus \mathsf{let}_d \vdash \varphi : \mathsf{mon}(e)^{p}}{\Gamma \vdash d(\bar{x}) : \mathsf{mon}(e)^{p}}\,\mathbb{E}^{\mathsf{let}}_{\mathsf{mon}}$$

$$\frac{e \notin \mathsf{events}(\varphi)}{\Gamma \vdash \varphi : \mathsf{mon}(e)^{p}}\,\mathsf{occ}^{p}_{\mathsf{mon}} \qquad \frac{\Gamma \vdash \varphi : \mathsf{mon}(e)^{\neg p}}{\Gamma \vdash \neg\varphi : \mathsf{mon}(e)^{p}}\,\neg^{p}_{\mathsf{mon}}$$

$$\frac{\Gamma \vdash \varphi : \mathsf{mon}(e)^{p} \quad \Gamma \vdash \psi : \mathsf{mon}(e)^{p}}{\Gamma \vdash \varphi \wedge \psi : \mathsf{mon}(e)^{p}}\,\wedge^{p}_{\mathsf{mon}} \qquad \frac{\Gamma \vdash \varphi : \mathsf{mon}(e)^{p}}{\Gamma \vdash \exists x.\ \varphi : \mathsf{mon}(e)^{p}}\,\exists^{p}_{\mathsf{mon}}$$

$$\frac{\Gamma \vdash \varphi : \mathsf{mon}(e)^{p}}{\Gamma \vdash \bigcirc\,\varphi : \mathsf{mon}(e)^{p}}\,\bigcirc^{p}_{\mathsf{mon}} \qquad \frac{\Gamma \vdash \varphi : \mathsf{mon}(e)^{p}}{\Gamma \vdash \bullet\,\varphi : \mathsf{mon}(e)^{p}}\,\bullet^{p}_{\mathsf{mon}}$$

$$\frac{\Gamma \vdash \varphi : \mathsf{mon}(e)^{p} \quad \Gamma \vdash \psi : \mathsf{mon}(e)^{p}}{\Gamma \vdash \varphi\,\mathsf{S}\,\psi : \mathsf{mon}(e)^{p}}\,\mathsf{S}^{p}_{\mathsf{mon}} \qquad \frac{\Gamma \vdash \varphi : \mathsf{mon}(e)^{p} \quad \Gamma \vdash \psi : \mathsf{mon}(e)^{p}}{\Gamma \vdash \varphi\,\mathsf{U}\,\psi : \mathsf{mon}(e)^{p}}\,\mathsf{U}^{p}_{\mathsf{mon}}$$

$$\frac{\Gamma, \mathsf{let}_d : \varphi \vdash \psi : \mathsf{mon}(e)^{p}}{\Gamma \vdash \mathsf{let}\,d(\bar{x}) = \varphi\,\mathsf{in}\,\psi : \mathsf{mon}(e)^{p}}\,\mathsf{let}^{p}_{\mathsf{mon}}$$

Figure 3.1: MFOTL (Anti) Monotonicity Rules [12, Appendix A]

3.3.2 CRE[MFOTL] Refinement

In this subsection we recall refinement of Lex specifications as it is described in Hublet et. al [12]. We then go ahead and proof that whenever a CRE specification refines another CRE specification, their compiled MFOTL formulas are likewise refinements of each other.

Hublet et al. [12] use the rules in Figure 3.3 to define refinement for CRE[MFOTL] specifications in the following manner.

Definition 3.32 *Refinement of CRE specifications is defined by the rules in Figure 3.3. A specification S_C refines a specification S_A, written as $S_C \sqsubseteq_\pi S_A$ under a refinement mapping π iff there exist Γ_C and Γ_A such that $S_C, \Gamma_C \rightsquigarrow_\pi S_A, \Gamma_A$.*

The refinement rules in Figure 3.3 depend on the concept of monotonic and anti-monotonic predicates. Figure 3.1 shows the monotonicity rules for MFOTL predicates, in which $p, q \in \{+, -\}$, with $\neg+ := -$ and $\neg- := +$. A predicate P appears monotonically (or anti-monotonically) in a formula φ, if the addition of an event $P(\bar{x})$ cannot make an already compliant (or non-compliant) trace non-compliant (or compliant). For example in the formula $\varphi := \square(\neg p(1))$ the predicate P does not appear monotonically, because the addition of an event at any time $P(1)$ would make a trace that was previously compliant with φ, non-compliant with φ, on the other hand P appears anti-monotonically in φ. It is important to note that a predicate can appear monotonically, anti-monotonically, both monotonically and anti-

$$\frac{\exists r : \varphi \Rightarrow p \in C \qquad \forall r : \varphi \Rightarrow p \in C.\ \Gamma \vdash^{\Phi} \varphi : \mathsf{mon}(e)^{q} \textbf{ and } \Gamma \vdash^{\Phi} \mathsf{exc}(r) : \mathsf{mon}(e)^{\neg q}}{\Gamma \vdash^{\Phi} p(\overline{x}) : \mathsf{mon}(e)^{q}}\ \textsc{con}^{q}_{\mathsf{mon}}$$

$$\frac{\forall r : \varphi \Rightarrow \mathsf{exc}(s) \in E.\ \Gamma \vdash^{\Phi} \varphi : \mathsf{mon}(e)^{p} \textbf{ and } \Gamma \vdash^{\Phi} \mathsf{exc}(r) : \mathsf{mon}(e)^{\neg p}}{\Gamma \vdash^{\Phi} \mathsf{exc}(s)(\overline{x}) : \mathsf{mon}(e)^{p}}\ \textsc{exc}^{p}_{\mathsf{mon}}$$

$$\frac{r : \varphi \Rightarrow \psi \in R \quad \Gamma \vdash \varphi \rightarrow \psi : \mathsf{mon}(e)^{p} \quad \Gamma \vdash \mathsf{exc}(r) : \mathsf{mon}(e)^{p}}{\Gamma \vdash^{\Phi} \mathsf{comply}(r) : \mathsf{mon}(e)^{p}}\ \textsc{reg}^{p}_{\mathsf{mon}}$$

$$\frac{r : \varphi \Rightarrow \psi \in R}{\Gamma \vdash (C, R, E) : \mathsf{mon}(\mathsf{exc}(r))^{+}} \qquad \frac{r : \varphi \Rightarrow p \in C \quad \Gamma \vdash (C, R, E) : \mathsf{mon}(p)^{\neg q}}{\Gamma \vdash (C, R, E) : \mathsf{mon}(\mathsf{exc}(r))^{q}}$$

$$\frac{r : \varphi \Rightarrow \mathsf{exc}(s) \in E \quad \Gamma \vdash (C, R, E) : \mathsf{mon}(\mathsf{exc}(s))^{\neg p}}{\Gamma \vdash (C, R, E) : \mathsf{mon}(\mathsf{exc}(r))^{p}}$$

$$\frac{\forall r : \varphi \Rightarrow \psi \in R.\ \Gamma \vdash^{(C,R,E)} \mathsf{comply}(r) : \mathsf{mon}(e)^{p}}{\Gamma \vdash (C, R, E) : \mathsf{mon}(e)^{p}}$$

Figure 3.2: CRE (Anti) Monotonicity Rules [12, Appendix A]

monotonically, or neither monotonically nor anti-monotonically in a formula.

Figure 3.4 shows an algorithm that converts a concrete MFOTL trace to an abstract one, according to a formula φ that defines an event $e(x_1, \ldots, x_k)$ in terms of the concrete events $e_1, \ldots, e_\ell$. This function does for MFOTL traces, what the mediator function π from Definition 3.32 and its extended form $\vec{pi}$ from Definition 3.29 do for system traces.

Definition 3.33 *Let* MAP *be a function as defined in Figure 3.4. Then the mediator function* $\mathsf{d}(\textsc{map})$ *is defined as follows:*

$$\begin{aligned} \mathsf{d}(\textsc{map})(\sigma_{\mathbf{s}}, \mathsf{tick}) &:= \mathsf{tick} \\ \mathsf{d}(\textsc{map})(\sigma_{\mathbf{s}}, \mathsf{skip}) &:= \mathsf{skip} \\ \mathsf{d}(\textsc{map})(\sigma_{\mathbf{s}}, D_C) &:= D_A \end{aligned}$$

where D_A is defined as the database of the last time point of the trace being returned by the MFOTL trace $\textsc{map}(\mathsf{c}(\sigma_{\mathbf{s}} \cdot D))$.

Lemma 3.34 *For any MFOTL trace $\sigma \in \mathbb{T}$, and map function* MAP

$$\overrightarrow{\mathsf{d}(\textsc{map})}(\mathsf{c}(\sigma)) = \mathsf{d}(\textsc{map}(\sigma))$$

The following theorem states that any valid refinement of an enforceable CRE[MFOTL] specification is itself enforceable.

$$\varphi \Updownarrow^{+} \psi := \varphi \Longrightarrow \psi \qquad \varphi \Updownarrow^{-} \psi := \varphi \Longleftarrow \psi$$

$$\frac{}{S, \Gamma \rightsquigarrow_{\mathrm{id}} S, \Gamma}\ \textsc{refl} \qquad \frac{S_1, \Gamma_1 \rightsquigarrow_{\pi_1} S_2, \Gamma_2 \quad S_2, \Gamma_2 \rightsquigarrow_{\pi_2} S_3, \Gamma_3}{S_1, \Gamma_1 \rightsquigarrow_{\pi_1 \circ \pi_2} S_3, \Gamma_3}\ \textsc{trans}$$

$$\frac{\begin{array}{c} C' = C \cup \{r : \varphi \Rightarrow e(x_1, \ldots, x_k)\} \quad \mathsf{events}(\varphi) \setminus \mathbb{E} = \{e_1, \ldots, e_\ell\} \\ \mathbb{E}' = \mathbb{E} \setminus \{e\} \cup \{e_1, \ldots, e_\ell\} \quad \iota' = \iota[i.\ e_i \mapsto k_i] \setminus \{e\} \\ \kappa' = \kappa[i.\ e_i \mapsto \tau_i] \setminus \{e\} \quad \Gamma' = \Gamma[i.\ e_i \mapsto \tau_i'] \setminus \{e\} \\ \forall i.\ \tau_i' \sqsubseteq \tau_i \quad \Gamma' \vdash \varphi : \Gamma(e) \quad \nexists r', \varphi', \overline{x}'.\ r' : \varphi' \Rightarrow e(\overline{x}') \in C \\ \pi = \mathrm{MAP}_{e(x_1,\ldots,x_k),\varphi,\{e_1,\ldots,e_\ell\}} \end{array}}{((\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota), (C, R, E), \kappa), \Gamma \rightsquigarrow_{\pi} ((\mathbb{D}, \mathbb{E}', \mathbb{F}, \iota'), (C', R, E), \kappa'), \Gamma'}\ \textsc{ref}$$

$$\frac{\begin{array}{c} R' = R \cup \{r : \varphi' \Rightarrow \psi'\} \setminus \{r : \varphi \Rightarrow \psi\} \\ \varphi' \to \psi' \Longrightarrow \varphi \to \psi \quad \Gamma \vdash^{(C,R',E)} \mathsf{comply}(r) : \mathbb{C} \end{array}}{((\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota), (C, R, E), \kappa), \Gamma \rightsquigarrow_{\mathrm{id}} ((\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota), (C, R', E), \kappa), \Gamma}\ \textsc{imp}_R$$

$$\frac{\begin{array}{c} C' = C \cup \{r : \varphi' \Rightarrow p\} \setminus \{r : \varphi \Rightarrow p\} \quad R' = R \cup \{s : \psi \Rightarrow \chi\} \\ q \in \{+, -\} \quad \Box(\psi \to \chi) \Longrightarrow \varphi' \Updownarrow^{q} \varphi \\ (C \setminus \{r : \varphi \Rightarrow p\}, R', E) : \mathsf{mon}(p)^q \quad \Gamma \vdash^{(C',R,E)} \mathsf{comply}(s) : \mathbb{C} \\ \forall \tau \in \{\mathbb{C}, \mathbb{S}\}.\ \Gamma \vdash^{(C,R,E)} p : \tau \Longrightarrow \Gamma \vdash^{(C',R,E)} p : \tau \end{array}}{((\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota), (C, R, E), \kappa), \Gamma \rightsquigarrow_{\mathrm{id}} ((\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota), (C', R', E), \kappa), \Gamma}\ \textsc{imp}_C$$

$$\frac{\begin{array}{c} E' = E \cup \{r : \varphi' \Rightarrow \mathsf{exc}(s)\} \setminus \{r : \varphi \Rightarrow \mathsf{exc}(s)\} \quad R' = R \cup \{s : \psi \Rightarrow \chi\} \\ q \in \{+, -\} \quad \Box(\psi \to \chi) \Longrightarrow \varphi' \Updownarrow^{q} \varphi \\ (C, R', E \setminus \{r : \varphi \Rightarrow \mathsf{exc}(s)\}) : \mathsf{mon}(\mathsf{exc}(s))^q \quad \Gamma \vdash^{(C,R',E')} \mathsf{comply}(s) : \mathbb{C} \\ \forall \tau \in \{\mathbb{C}, \mathbb{S}\}.\ \Gamma \vdash^{(C,R,E)} \mathsf{exc}(s) : \tau \Longrightarrow \Gamma \vdash^{(C,R,E')} \mathsf{exc}(s) : \tau \end{array}}{((\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota), (C, R, E), \kappa), \Gamma \rightsquigarrow_{\mathrm{id}} ((\mathbb{D}, \mathbb{E}, \mathbb{F}, \iota), (C, R', E'), \kappa), \Gamma}\ \textsc{imp}_E$$

Figure 3.3: CRE[MFOTL] refinement rules [12, Figure 7]

Theorem 3.35 *If $S_C \sqsubseteq_\pi S_A$ under a refinement mapping and S_A is enforceable, then S_C is enforceable.*

Theorem 3.36 *If $S_C \sqsubseteq_\pi S_A$, S_A is enforceable, and $\sigma \vDash S_C$, then $\pi(\sigma) \vDash S_A$.*

Hublet et al. [12] have proved Theorem 3.35 [12, Theorem 4] and Theorem 3.36 [12, Theorem 3].

Theorem 3.37 *If $S_C \sqsubseteq_\pi S_A$ and S_A is enforceable then there exist two enforcement event systems Ξ_C and Ξ_A such that they are sound for S_C and S_A respectively, and*

$$
\begin{array}{l}
\textbf{let } \mathsf{map}_{e(x_1,\ldots,x_k),\varphi,E}\ \sigma = \\
\quad \langle(\tau_i, D_i)_{i\in\mathbb{N}}\rangle \leftarrow \sigma \\
\quad \textbf{for } i \in \mathbb{N} \textbf{ do} \\
\quad\quad D_i' \leftarrow D_i \cup \{e(v(x_1),\ldots,v(x_k)) \mid v, i \vDash_\sigma \varphi\} \\
\quad\quad D_i' \leftarrow D_i' \setminus \{e(\overline{d}) \mid (d_1,\ldots,d_{\iota(e)}) \in \mathbb{D}^{\iota(e)}, e \in E\} \\
\quad \textbf{end for} \\
\quad \textbf{return } \langle(\tau_i, D_i')_{i\in\mathbb{N}}\rangle
\end{array}
$$

Figure 3.4: MAP function used in Figure 3.3

refine each other via the mediator function $\mathsf{d}(\pi)$*, i.e.*

$$\Xi(\mathcal{E}_C) \sqsubseteq_{\mathsf{d}(\pi)} \Xi(\mathcal{E}_A).$$

Proof for Theorem 3.37 Let $S_C \sqsubseteq_\pi S_A$, and S_A be enforceable.

By the enforceability of S_A we know that there exists an enforcer $\mathcal{E}_A = (\mathcal{S}_A, s_0, \mu_A)$ which is sound for S_A. By Theorem 3.35 we also know that there exists an enforcer $\mathcal{E}_C = (\mathcal{S}_C, s_0, \mu_C)$ which is sound for S_C.

Since $S_C \sqsubseteq_\pi S_A$ there exist two contexts Γ_C and Γ_A such that $S_A, \Gamma_A \rightsquigarrow_\pi S_C, \Gamma_C$. This proof proceeds by induction over the different CRE[MFOTL] refinement rules for $\rightsquigarrow_\pi$. We show by induction on $\rightsquigarrow_\pi$ that when $S_A, \Gamma_A \rightsquigarrow_\pi S_C, \Gamma_C$ and $\exists \mathcal{E}_A. \forall \sigma \in \mathbb{T}. \sigma \in \mathcal{L}(S_A)$ then there exist two enforcement event systems Ξ_C and Ξ_A such that Ξ_C is sound for S_C, Ξ_A is sound for S_A, and $\Xi_C \sqsubseteq_{\mathsf{d}(\pi)} \Xi_A$.

In the rest of this proof, let $R_\pi \subseteq \mathbb{T}_{\mathbf{s}C} \times \mathbb{T}_{\mathbf{s}A}$ be the simulation relation defined by $\overrightarrow{\mathsf{d}(\pi)} : \mathbb{T}_{\mathbf{s}C} \to \mathbb{T}_{\mathbf{s}A}$.

REFL In this case $S_C = S_A$ and π is the identity mapping. By Theorem 3.25 we know that $\Xi(\mathcal{E}_A)$ is sound for S_A. Finally, any enforcement event system $\Xi(\mathcal{E}_A)$ is trivially a refinement of itself under the identity mapping (and the identity simulation relation).

TRANS For this case we can simply apply the induction hypothesis twice.

REF We know that two enforcers $\mathcal{E}_C$ that is sound for S_C exists (Theorems 3.35) and that its induced event Systems are also sound for S_C (3.25).

It remains to be shown that the two induced event systems refine each other.

Let's consider the situation in point 2 of Definition 3.28. We have two concrete system traces $\sigma_{\mathbf{s}C}, \sigma'_{\mathbf{s}C} \in \mathbb{T}_{\mathbf{s}C}$, one abstract system trace $\sigma_{\mathbf{s}A} \in \mathbb{T}_{\mathbf{s}A}$, and a concrete event $e_C \in \mathfrak{E}_C$ such that $\sigma_{\mathbf{s}C} \xrightarrow{e_C}_C \sigma'_{\mathbf{s}C}$. And if this is the case we want to prove that there exists an abstract system trace $\sigma'_{\mathbf{s}A} \in \mathbb{T}_{\mathbf{s}A}$ such that $\sigma_{\mathbf{s}A} \xrightarrow{\mathsf{d}(\pi)(\sigma_{\mathbf{s}C}, e_C)}_A \sigma'_{\mathbf{s}A}$.

We can argue that the abstract enforcement event system can iteratively be defined by applying the mediator function $\mathsf{d}(\pi)$ to all traces and transitions of the concrete enforcement event system. A transition $\sigma_{\mathbf{s}C} \xrightarrow{e_C}_C \sigma'_{\mathbf{s}C}$ would then become $\overrightarrow{\mathsf{d}(\pi)}(\sigma_{\mathbf{s}C}) \xrightarrow{\mathsf{d}(\pi)(\sigma_{\mathbf{s}C}, e_C)}_A \overrightarrow{\mathsf{d}(\pi)}(\sigma'_{\mathbf{s}C})$. And the simulation relation R can be defined by the aforementioned R_π. We call the so constructed enforcement event system Ξ_A. The event system Ξ_A is sound for S_A because of Theorem 3.36.

IMP$_R$ This rule simply strengthens a policy and thus reduces the set of "allowed" traces. By Theorem 3.35 we know that there exists an enforcer $\mathcal{E}_C$ that is sound for S_C. By the fact that $\mathcal{L}(S_C) \subseteq \mathcal{L}(S_A)$ we can deduce that $\mathcal{E}_C$ is also sound for S_A. And thus the induced event system for $\mathcal{E}_C$ is sound for both S_C and S_A, and also a refinement of itself.

IMP$_C$ and **IMP**$_E$ Without loss of generality we will only consider the case of IMP$_C$. The case of IMP$_E$ is analogous. This rule strengthens or weakens the definition of an internal event alongside the addition of a new, enforceable regulative rule. The regulative rule, like in the IMP$_R$ case, reduces the set of "allowed" traces. The strengthened or weakened definition has the effect that the defined event will occur less or more often, but under the (anti) monotonicity constraints it is ensured that the missing or extra occurrences of the event cannot affect compliance.

The event systems Ξ_C and Ξ_A can be simply constructed by inducing them from the sound enforcers $\mathcal{E}_C$ and $\mathcal{E}_A$ which we know must exist. Then we can build up the simulation relation R, by iteratively following all possible transitions and adding the created trace pairs to R. e.g. starting with the empty trace, we check what trace the concrete enforcement event system Ξ_C would produce when the event e occurs and what trace the abstract enforcement event system Ξ_A would produce when the event e occurs, and then we add the pair of resulting traces to R. $\square$

Chapter 4

Implementation

4.1 Introduction

This chapter explores the changes and additions to the Lex language to include CRE[MFOTL] refinement from Chapter 3. With refinement in Lex, an existing and possibly fairly abstract legal formalization can be refined to more closely match a concrete system. First we introduce the new language features introduced for refinement, and then we explore those features in more detail by refining the EU ePrivacy Directive example in Lex.

4.2 Refinement of Lex Specifications

Refinement introduces a number of new statements in Lex. Lex refinement files use the file extension `.rex`. For refinement, we made the following additions to Lex.

- **Reference to the abstract Lex specification** (Keyword: `refine`) Any Lex refinement specification will start with a `refine` statement. This statement is simply used to specify the underlying Lex specification that is being refined. Syntactically it works the same way as an `import` statement (e.g. `refine` edp).
- **Event Refinement** There are two ways to refine non-internal events in Lex. First an event can be marked as assumed to be always true or always false, and second an event can be refined by a combination of other events. In the second case, a previously non-internal event becomes an internal event in the refined specification. Section 4.2.1 dives further into the details of event refinement.
- **Rule strengthening and weakening** (Keyword: `replace`) With refinement, old rules in a Lex specification can be replaced by new rules in accordance with the implication rules for CRE[MFOTL] refinement

from Figure 3.3. Section 4.2.2 contains more details on the specifics of rule replacement.

- **Type Refinements** (Structure: `refine type` ”abstract type” `is` ”concrete type”) Type refinements are used to specify concrete data types for types that might have been left abstract in the original Lex specification. For example, it might be unknown and not necessary to know in a legal formalization whether a person is represented as a string or integer identifier, but in the refinement specification we do know which data type is used by the target system.
- **Additional obligations and constitutive rules** A refined specification can always contain additional obligations that are not explicitly used to replace old obligations. Similarly, a refined specification can contain new constitutive rules that define events which may be used anywhere in the refined specification. For reasons of simplicity, we have opted to disallow the introduction of new exceptions in refined specifications, but exception rules can still be part of a refinement specification as long as they are explicitly used to replace existing exceptions.

4.2.1 Event Refinement

Refinement rules work almost exactly like constitutive rules. The main difference is that the refined event in a refinement rule is not an internal event like the constituted event in a constitutive rule, but it is a non-internal event that gets internalized by the refinement. The following example shows how this can be expressed as a refinement rule. Note the use of the `refine` keyword to indicate that this is a refinement rule.

```
rule
  whenever
    consentByEmail(u,s) OR consentByButton(u,s)
  refine
    consent(u,s)
```

In refinement the conditions must have at least the same capability as the refined event.

Assumptions are in essence syntactic shortcuts for refinement rules that have the condition `true` or `false`.

```
assume true objectToSimilarEmailMarketing
```

means that we assume that every user `u` has objected to receiving marketing emails from the sender `s` about products or services in category `c`. And this assumption could also be written as

```
rule
  whenever
    true
  refine
    objectToSimilarEmailMarketing(u, s, c)
```

4.2.2 Rule Strengthening and Weakening

Replacement rules are structured and written very similarly to other Lex rules. They consist of a reference to an old rule and one or more references to new rules replacing the old one. Refinement rules have the form

```
replace
  (strengthen | weaken)
    [old rule reference]
  by
    [new rule reference(s)]
```

Rule replacement is the Lex equivalent to the implication (IMP) rules from CRE[MFOTL] in Figure 3.3.

Strengthening Obligations One obligation rule can be replaced by one or more obligations that are at least as strong as the old obligation. Any exception that applied to the old obligation will directly apply to the new obligations. Weakening of obligations is not allowed, because then the refined specification would not correctly implement the old one. This corresponds to the IMP_R rule in Figure 3.3.

Replacing Constitutive Rules A single constitutive rule can be replaced by a single constitutive rule and zero or more obligations. Any exception to the old constitutive rule will apply to the new constitutive rule, but not the new obligations. This corresponds to the IMP_C rule in Figure 3.3. A constitutive rule can only be strengthened, if the constitutive event appears monotonically in the old Lex specification, and it can only be weakened if the constitutive event appears anti-monotonically in the old Lex specification. This is condition is necessary, because strengthening a constitutive rule means that under the stronger rule, the constituted event might occur less frequently and under a weaker rule it might occur more frequently. The (anti) monotonicity condition guarantees that, what in essence is the removal (or addition) of event occurrences in the refined specification, cannot affect whether a trace complies with a specification or not.

Replacing Exceptions Analogous to the replacement of constitutive rules, a single exception can be replaced by a single exception and zero or more

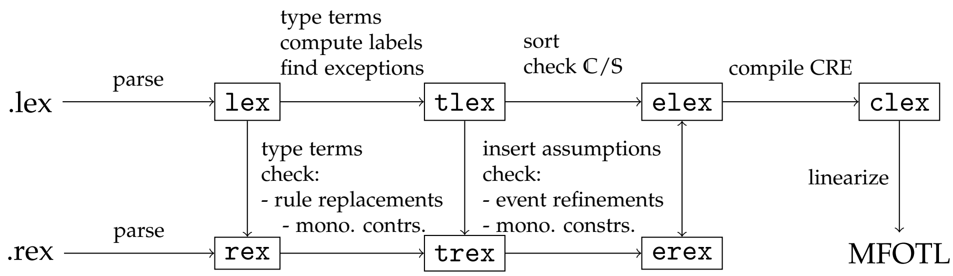

Figure 4.1: Structure of the Lex compiler [12, Figure 13]

obligations. This is analogous to the $\textsc{imp}_E$ rule in Figure 3.3. Like with constitutive rules, the internally created exception event, must appear either monotonically or anti-monotonically in the old Lex specification such that it can be strengthened or weakened respectively.

4.3 Implementation in the Lex Compiler

We implemented these changes and additions to Lex in the Lex compiler [1]. The changes amount to about 900 lines of OCaml code. The additions are mainly concentrated in the two files `src/rtyping.ml` and `src/refinement.ml`. The structure of the Lex compiler is shown in Figure 4.1.

The step from `rex` to `trex` is implemented in the `src/rtyping.ml` file. In this step terms get type checked, i.e. arguments to functions and event parameters. Also section labels get computed. Finally, the replacement of rules gets checked on the criteria explained in Section 4.2.2. However, monotonicity depends on the topological ordering, it can only be determined whether an event appears monotonically or anti-monotonically in the Lex specification when all its occurrences in the specification are known. Therefore, a set of monotonicity constraints gets computed in this step and forwarded to the next step.

The step from `trex` to `erex` is implemented in the `src/refinement.ml` file. During this step newly refined events get internalized and the capabilities of their new definitions get checked. Assumptions get unfolded into refinement rules and. The forwarded monotonicity constraints get checked based on the topological order. Also in this step, the ”inherited” exceptions from replaced rules get correctly mapped to the new rules.

[1] Lex Compiler https://gitlab.inf.ethz.ch/OU-BASIN/lex/

Chapter 5

Case Study

5.1 Refinement of the EU ePrivacy Directive in Lex

5.1.1 Refinement of a Lex Specification

Figure 5.1 shows one possible refinement of the Lex specification from Figure 2.11. This Lex refinement refines the meaning of prior consent via a refinement rule. This refinement rule is more closely modeled after how

```
refine epd

refine type person is string

observable event userGivesConsent
    """person {s} gives consent to receive automated marketing communication from person {r}"""
    r: person
    s: person

observable event userWithdrawsConsent
    """person {s} withdraws consent to receive automated marketing communication from person {r}"""
    r: person
    s: person

rule "refine marketing consent"
    whenever
        (NOT userWithdrawsConsent(r, s)) SINCE userGivesConsent(r, s)
    refine
        marketingConsent(r, s)

assume true objectToSimilarEmailMarketing
assume true isNaturalPerson
assume false fax
assume false automatedCall

observable event over18
    """person {r} is over 18 years old"""
    r: person

rule "stronger_consent"
    whenever
        automatedCommunication(s, r, "marketing", c)
    oblige
        marketingConsent(r, s)
        over18(r)
    enforceable suppressing automatedCommunication

replace
    strengthen
        article "13" paragraph "1"
    by
        rule "stronger_consent"
```

Figure 5.1: Lex Refinement of Figure 2.11

a concrete system might implement what it means that a user has given consent at a certain point in time. Specifically, it combines two new concrete events, one for giving and for withdrawing consent.

The type for the data type of person, which was left open in the original Lex specification, is declared as a string.

It is assumed that all users are natural persons, and out of caution it is also assumed that all users have objected to receiving marketing about related products.

The fax and automated call events are assumed to never occur, which would represent many modern systems that do not even have these a fax or automated phone attached.

An event is introduced to represent that a user is over 18 years old. And the obligation rule about receiving unsolicited marketing that disallows sending any marketing emails to users under 18, regardless of whether they have given consent.

5.1.2 Pretix

Pretix [2] is an open source event ticketing system written in Python and built on Django. It offers products for ticket sales, wait lists, on location ticket sales, and ticket scanning. We are interested in the pre-event ticket sales part. We evaluated a refined Lex formalization of the email marketing regulations in the EU ePrivacy Directive on the Pretix platform.

Instrlib [11] is a python library for instrumenting Python programs for enforcement with the EnfGuard [9] enforcer. It has been designed to facilitate especially seamless instrumentation of Django applications. Instrlib takes over the interaction between EnfGuard and an instrumented program. During this thesis, while working on the case-study, we extended the logger in Instrlib with a cache. The cache has an adjustable timeout value, and serves the purpose that when the same event gets queried multiple times in quick succession, its corresponding values do not have to be recomputed every time, since they are unlikely to have changed. In some cases this can considerably reduce the delay that enforcement introduces.

For this, we modified Pretix and instrumented this version of it with Instrlib [1] [2]. The modifications to Pretix consist of new fields to the customer model indicating consent to receive marketing emails, and modified views to give and withdraw this consent.

[1] Instrlib+Pretix case study https://gitlab.inf.ethz.ch/skrstic/proactive-enforcement-library/-/tree/refinement

[2] Instrumented Pretix fork https://github.com/jnzd/pretix

Pretix stores customer emails in the order data model and the customer data model for users that create an account. We have instrumented the email fields of these two model classes, such that accessing them might get suppressed if needed. Further we annotated the function for sending emails with the ″marketing″ purpose and we annotated our newly added consent checkbox in such a way that updating the marketing consent value, triggers an event that gets reported to Instrlib and the enforcer.

Pretix is a large and feature rich solution for ticket sales and event organization. It has other aspects that might be interesting for enforcement testing in the future, such as its wait list, which might fall under data minimization requirements.

5.2 Lessons Learned

Formalizing any regulation in Lex requires a good understanding of the intended purpose of the regulation and a certain amount of creativity in how to model which concepts. In this case study, the addition of refinement allowed us to move certain complexities, such as the implementation of consent, out of the original Lex specification and to move them into the refinement file. Even with a relatively small example, our approach appears to be highly promising, and its advantages should only become more apparent in larger projects.

Another aspect that has become clear is that in order to write a good and useful refinement mapping, a good level of understanding of the underlying code base is required. One must know which actions are performed where and how. To implement the actual interaction with the enforcer, an integrated solution like Instrlib has proved very helpful.

During basic testing of the instrumented Pretix code base, one tricky challenge has been not to instrument too much. If a part of the code gets instrumented that is frequently accessed, every access must go through the enforcer, which can quickly add very long delays. We added a cache to Instrlib to mitigate this issue, but it remains that the fewer parts of the code that have to be instrumented to successfully enforce a policy, the less noticeable the presence of the enforcer becomes. Determining which parts must always be logged and which parts can be safely ignored (sometimes) without losing soundness.

Chapter 6

Conclusion

Motivated by the need for reusable and adaptable legal specifications in runtime enforcement, this thesis presents the addition of refinement to Lex. Refinement in Lex is a flexible and powerful mechanism to adapt and reuse an existing Lex specification for different use cases. Our refinement rules for Lex ensure That a valid refinement of an enforceable Lex specification is also enforceable.

In Chapter 3 we have introduced our refinement rules, and we have connected them to the long-standing concept of specification refinement. Specifically we have shown it to be analogous to the refinement notion for event systems used by Sprenger et al. [17]. For this purpose, we have modeled proactive enforcement as an event system with labeled transitions. In Chapter 4 we have implemented the refinement rules for the CRE[MFOTL] logic introduced in Chapter 3 in Lex. Finally, Chapter 5 presents a case study that demonstrates a possible use case for refinement in Lex based on the ePrivacy Directive [1] and the Pretix ticketing system [2].

To our knowledge, Lex is the first specification language for legal norms that supports an inheritance-like feature such as refinement.

In the future Lex refinement could be extended to be even more flexible in what it allows, like defining new global exceptions or step-wise refinement, where multiple Lex refinement specifications iterate on each other. Stepwise refinement should be a relatively simple change to the Lex compiler. This could allow for interesting use cases, one person (or more likely group of people) could formalize and carefully audit a law like the GDPR, another could refine it for a specific class of applications, say social media apps, and then finally any social media provider could refine the already refined specification even further. An ecosystem of downloadable Lex specifications, like package management in general purpose programming languages, could be built around Lex and Lex refinement.

One important challenge that Lex and Lex refinement do not solve entirely is the issue of correctness. Lex does provide logical correctness, i.e. it ensures that a type correct Lex specification is enforceable under the given capabilities for events, it also ensures that enforceability is transitive through its refinement system. But what Lex does not and cannot ensure is that a Lex specification correctly captures the intended meaning of a legal norm. It also cannot ensure that a refinement maintains and does not change the semantic meaning of a formalization. Lex attempts to make auditing a formalization as convenient as possible, thus reducing the error probability in specifications, but this does require careful evaluation by humans and more likely, legal experts. With the continued rise of artificial intelligence and large language models, it might become easier to automate the extraction of the logical structure of legal norms in an automated fashion. However, even such systems cannot currently ensure that a formalization correctly represents an underlying law automatically. This remains a large unsolved problem that may well be unsolvable as long as laws are written in natural language that is inherently imprecise. Both by necessity and design, Lex tries to give a readable method for representing the logical structure of legal norms which is subject to human interpretation.

Bibliography

[1] Directive 2002/58/ec of the European Parliament and of the Council.

[2] Pretix – Reinventing ticket sales for conferences, festivals, exhibitions, ... https://pretix.eu/about/en/.

[3] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[4] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. In *Runtime Verification: 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings 4*, pages 40–58. Springer, 2013.

[5] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM (JACM)*, 62(2):1–45, 2015.

[6] Gerhard Brewka and Thomas Eiter. Prioritizing default logic. In *Intellectics and Computational Logic: Papers in Honor of Wolfgang Bibel*, pages 27–45. Springer, 2000.

[7] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.

[8] Jonas Degelo. A language for formalizing legal provisions. Semester project at ETH Zürich, Fall 2024, 2024.

[9] François Hublet, Leonardo Lima, David Basin, Srđan Krstić, and Dmitriy Traytel. Proactive real-time first-order enforcement. In *International Conference on Computer Aided Verification*, pages 156–181. Springer, 2024.

[10] François Hublet, Leonardo Lima, David Basin, Srđan Krstić, and Dmitriy Traytel. Scaling Up Proactive Enforcement. In Ruzica Piskac and Zvonimir Rakamaric, editors, *International Conference on Computer Aided Verification*. Springer, 2025. to appear.

[11] François Hublet, Linda Hu, David Basin, and Srđan Krstić. Instrumenting runtime enforcement. Technical report, ETH Zürich, Department of Computer Science, 2025.

[12] François Hublet, Jakob Merane, Jonas Degelo, Linda Hu, David Basin, and Srđan Krstić. Lex: A Tool-Supported Methodology for Compliance by Design. 2025. Working draft.

[13] Filippo Lancieri. Narrowing data protection's enforcement gap. *Me. L. Rev.*, 74:15, 2022.

[14] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.

[15] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: a programming language for the law. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, August 2021.

[16] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[17] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A Wolf, Peter Müller, Martin Clochard, and David Basin. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020.

[18] L Thomas Van Binsbergen, Lu-Chi Liu, Robert Van Doesburg, and Tom Van Engers. eflint: a domain-specific language for executable norm specifications. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 124–136, 2020.

[19] Sheila Zingg, Srđan Krstić, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. Verified first-order monitoring with recursive rules. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 236–253. Springer, 2022.

Compiled Lex Examples

The compiled output of the ePrivacy Directive formalization in Lex (Figure 2.11) is shown in Figure 1 and Figure 2. The compilation of the refined specification from Figure 5.1 is shown in Figure 3 and Figure 4.

```
automatedCall(r: int, s: int, p: string, c: string)+-
email(r: int, s: int, p: string, c: string)+-
fax(r: int, s: int, p: string, c: string)+-
isNaturalPerson(s: int)
marketingConsent(r: int, s: int)
objectToSimilarEmailMarketing(r: int, s: int, c: string)
obtainedEmailForCategory(r: int, s: int, c: string)
```

Figure 1: Compiled MFOTL Signature of the Lex Specification

```
LET automatedCommunication(_p1 : int, _p2 : int, _p3 : string
                                                 _p4 : string)- =
    email(_p1, _p2, _p3, _p4) ∨ fax(_p1, _p2, _p3, _p4)
    ∨ automatedCall(_p1, _p2, _p3, _p4) IN
LET Exception4(c : string, r : int, s : int) =
    obtainedEmailForCategory(r, s, c)
    ∧ ¬objectToSimilarEmailMarketing(r, s, c) IN
LET Scope5(r : int) = isNaturalPerson(r) IN
□[0s,∞) (
    ∀c. ∀r. ∀s. automatedCommunication(s, r, "marketing", c
        ∧:L ¬Exception4(c, r, s) ∧ Scope5(r)
    →:L marketingConsent(r, s))
```

Figure 2: Compiled MFOTL formula of the Lex Specification

```
email(r: string, s: string, p: string, c: string)+-
obtainedEmailForCategory(r: string, s: string, c: string)
over18(r: string)
userGivesConsent(r: string, s: string)
userWithdrawsConsent(r: string, s: string)
```

Figure 3: Compiled MFOTL Signature of the refined Lex Specification

```
LET automatedCall(_p11 : string, _p12 : string, _p13 : string,
                  _p14 : string) = ⊥ IN
LET fax(_p15 : string, _p16 : string, _p17 : string, _p18 : string)
    = ⊥ IN
LET isNaturalPerson(_p19 : string) = ⊤ IN
LET marketingConsent(_p9 : string, _p10 : string) =
    ¬userWithdrawsConsent(_p9, _p10)
    S[0s,∞) userGivesConsent(_p9, _p10) IN
LET objectToSimilarEmailMarketing(_p20 : string, _p21 : string,
                                  _p22 : string) = ⊤ IN
LET Exception4(c : string, r : string, s : string) =
    obtainedEmailForCategory(r, s, c)
    ∧ ¬objectToSimilarEmailMarketing(r, s, c) IN
LET automatedCommunication(_p5 : string, _p6 : string, _p7 :
                       string, _p8 : string)- =
    email(_p5, _p6, _p7, _p8) ∨ fax(_p5, _p6, _p7, _p8)
    ∨ automatedCall(_p5, _p6, _p7, _p8) IN
LET Scope5(r : string) = isNaturalPerson(r) IN
□[0s,∞) (
    ∀c. ∀r. ∀s. automatedCommunication(s, r, "marketing", c)
        ∧:L ¬Exception4(c, r, s) ∧ Scope5(r)
    →:L marketingConsent(r, s) ∧ over18(r))
```

Figure 4: Compiled MFOTL formula of the refined Lex Specification

Additional Proofs

Proof for Lemma 3.26

Proof Let $\sigma_\mathbf{s} \in \mathbb{T}_\mathbf{s}$ and $\tau \in \mathbb{N}$. The proof for Lemma 3.26 proceeds by structural induction over system traces $\sigma_\mathbf{s} \in \mathbb{T}_\mathbf{s}$.

- **Base Case** $\sigma_\mathbf{s} = \varepsilon_\mathbf{s}$

$$
\begin{aligned}
\mathsf{c}'(\tau, \sigma_\mathbf{s} \cdot D) &= \mathsf{c}'(\tau, \varepsilon_\mathbf{s} \cdot D) \\
&= \mathsf{c}'(\tau, D) \\
&= \varepsilon \cdot \mathsf{c}'(\tau + 0, D) \\
&\overset{3.2}{=} \varepsilon \cdot \mathsf{c}'(\tau + \#\mathsf{tick}(\varepsilon_\mathbf{s}), D) \\
&\overset{3.7}{=} \mathsf{c}'(0, \varepsilon_\mathbf{s}) \cdot \mathsf{c}'(\tau + \#\mathsf{tick}(\varepsilon_\mathbf{s}), D) \\
&= \mathsf{c}'(0, \sigma_\mathbf{s}) \cdot \mathsf{c}'(\tau + \#\mathsf{tick}(\sigma_\mathbf{s}), D)
\end{aligned}
$$

- **Induction Hypothesis**

  For any true suffix $\sigma'_\mathbf{s}$ of $\sigma_\mathbf{s}$ (i.e. $|\sigma'_\mathbf{s}| < |\sigma_\mathbf{s}|$)

$$
\mathsf{c}'(\tau, \sigma'_\mathbf{s} \cdot D) = \mathsf{c}'(\tau, \sigma'_\mathbf{s}) \cdot \mathsf{c}(\tau + \#\mathsf{tick}(\sigma'_\mathbf{s}), D).
$$

- **Induction Step**

  We make a case distinction based on $\sigma_\mathbf{s}$. Without loss of generality we can assume that there are no skip events in $\sigma_\mathbf{s}$.

– $\sigma_{\mathbf{s}} = \mathsf{tick}^k$

$$\begin{aligned}
\mathsf{c}'(\tau, \sigma_{\mathbf{s}} \cdot D) &= \mathsf{c}'(\tau, \mathsf{tick}^k \cdot D) \\
&\overset{3.7}{=} (\tau + k, D) \\
&= \varepsilon \cdot (\tau + k, D) \\
&\overset{3.7}{=} \mathsf{c}(\tau, \varepsilon_{\mathbf{s}}) \cdot (\tau + k, D) \\
&= \mathsf{c}(\tau, \mathsf{tick}^k) \cdot (\tau + k, D) \\
&= \mathsf{c}(\tau, \sigma_{\mathbf{s}}) \cdot (\tau + k, D) \\
&\overset{3.7}{=} \mathsf{c}(\tau, \sigma_{\mathbf{s}}) \cdot \mathsf{c}'(\tau + k, D) \\
&= \mathsf{c}(\tau, \sigma_{\mathbf{s}}) \cdot \mathsf{c}'(\tau + \#\mathsf{tick}(\sigma_{\mathbf{s}}), D)
\end{aligned}$$

– $\sigma_{\mathbf{s}} = \mathsf{tick}^k \cdot D' \cdot \sigma'_{\mathbf{s}}$

$$\begin{aligned}
\mathsf{c}'(\tau, \sigma_{\mathbf{s}} \cdot D) &= \mathsf{c}'(\tau, \mathsf{tick}^k \cdot D' \cdot \sigma'_{\mathbf{s}} \cdot D) \\
&\overset{3.7}{=} (\tau + k, D') \cdot \mathsf{c}'(\tau + k, \sigma'_{\mathbf{s}} \cdot D) \\
&\overset{IH}{=} (\tau + k, D') \cdot \mathsf{c}'(\tau + k, \sigma'_{\mathbf{s}}) \cdot \mathsf{c}'(\tau + k + \#\mathsf{tick}(\sigma'_{\mathbf{s}}), D) \\
&= (\tau + k, D') \cdot \mathsf{c}'(\tau + k, \sigma'_{\mathbf{s}}) \cdot \mathsf{c}'(\tau + \#\mathsf{tick}(\sigma_{\mathbf{s}}), D) \\
&\overset{3.7}{=} \mathsf{c}'(\tau, \mathsf{tick}^k \cdot D' \cdot \sigma'_{\mathbf{s}}) \cdot \mathsf{c}'(\tau + \#\mathsf{tick}(\sigma_{\mathbf{s}}), D) \\
&= \mathsf{c}'(\tau, \sigma_{\mathbf{s}}) \cdot \mathsf{c}'(\tau + \#\mathsf{tick}(\sigma_{\mathbf{s}}), D) \qquad \square
\end{aligned}$$

Proof for Lemma 3.27

Proof The proof for Lemma 3.27 proceeds by structural induction over finite system traces $\sigma_{\mathbf{s}} \in \mathsf{traces}_f(\Xi(\mathcal{E}), \varepsilon_{\mathbf{s}})$. Later we will make the argument why the property extends to inifinite system traces too. Without loss of generality we can assume that $\sigma_{\mathbf{s}} \in \mathsf{trim}(\mathbb{T}_{\mathbf{s}})$, because the conversion function c removes implicitly trims system traces. Let $\mathcal{E} = (\mathcal{S}, s_0, \mu)$ be an enforcer.

- **Base Case** $\sigma_{\mathbf{s}} = \varepsilon_{\mathbf{s}}$

$$\mathsf{c}(\sigma_{\mathbf{s}}) = \mathsf{c}(\varepsilon_{\mathbf{s}}) \overset{3.8}{=} \varepsilon = \mathcal{E}(\varepsilon)$$

  The final equality follows from Algorithm 1 which defines, that enforcing the empty trace returns the empty trace.

- **Induction Hypothesis** For all true prefixes $\sigma'_{\mathbf{s}}$ of $\sigma_{\mathbf{s}} \in \mathsf{traces}_f(\Xi(\mathcal{E}), \varepsilon_{\mathbf{s}})$ (i.e. $|\sigma'_{\mathbf{s}}| < |\sigma_{\mathbf{s}}|$) there exists a trace $\sigma' \in \mathbb{T}$ such that $\mathsf{c}(\sigma'_{\mathbf{s}}) = \mathcal{E}(\sigma')$.

- **Induction Step**

We make a case distinction based on the two possible transitions of enforcement event systems from Definition 3.19. The update functions δ and γ are defined as in Definition 3.22.

- $\sigma_{\mathsf{s}} = \sigma'_{\mathsf{s}} \cdot \delta(\sigma'_{\mathsf{s}}, D)$

$$\begin{aligned}
\mathsf{c}(\sigma_{\mathsf{s}}) &= \mathsf{c}(\sigma'_{\mathsf{s}} \cdot \delta(\sigma'_{\mathsf{s}}, D)) \\
&\overset{3.8}{=} \mathsf{c}'(0, \sigma'_{\mathsf{s}} \cdot \delta(\sigma'_{\mathsf{s}}, D)) \\
&\overset{3.26}{=} \mathsf{c}'(0, \sigma'_{\mathsf{s}}) \cdot \mathsf{c}'(\#\mathsf{tick}(\sigma'_{\mathsf{s}}), \delta(\sigma'_{\mathsf{s}}, D)) \\
&\overset{3.8}{=} \mathsf{c}(\sigma'_{\mathsf{s}}) \cdot \mathsf{c}'(\#\mathsf{tick}(\sigma'_{\mathsf{s}}), \delta(\sigma'_{\mathsf{s}}, D)) \\
&\overset{IH}{=} \mathcal{E}(\sigma') \cdot \mathsf{c}'(\#\mathsf{tick}(\sigma'_{\mathsf{s}}), \delta(\sigma'_{\mathsf{s}}, D)) \\
&\overset{3.7,3.14}{=} \mathcal{E}(\sigma') \cdot (\#\mathsf{tick}(\sigma'_{\mathsf{s}}), \delta(\sigma'_{\mathsf{s}}, D)) \\
&= \mathcal{E}(\sigma' \cdot (\#\mathsf{tick}(\sigma'_{\mathsf{s}}), D))
\end{aligned}$$

  The final step follows from the definition of the reactive update function δ by means of the enforcer update function μ and how μ is used by in lines 6 and 7 of Algorithm 1 to issue an RCom-command.

- $\sigma_{\mathsf{s}} = \sigma'_{\mathsf{s}} \cdot \gamma(\sigma'_{\mathsf{s}}) \cdot \mathsf{tick}$

$$\begin{aligned}
\mathsf{c}(\sigma_{\mathsf{s}}) &= \mathsf{c}(\sigma'_{\mathsf{s}} \cdot \gamma(\sigma'_{\mathsf{s}}) \cdot \mathsf{tick}) \\
&\overset{3.7}{=} \mathsf{c}(\sigma'_{\mathsf{s}} \cdot \gamma(\sigma'_{\mathsf{s}})) \\
&\overset{3.8}{=} \mathsf{c}'(0, \sigma'_{\mathsf{s}} \cdot \gamma(\sigma'_{\mathsf{s}})) \\
&\overset{3.26,\mathsf{skip}}{=} \mathsf{c}'(0, \sigma'_{\mathsf{s}}) \cdot \mathsf{c}'(\#\mathsf{tick}(\sigma'_{\mathsf{s}}, \gamma(\sigma'_{\mathsf{s}}))) \\
&\overset{3.8}{=} \mathsf{c}(\sigma'_{\mathsf{s}}) \cdot \mathsf{c}'(\#\mathsf{tick}(\sigma'_{\mathsf{s}}, \gamma(\sigma'_{\mathsf{s}}))) \\
&\overset{IH}{=} \mathcal{E}(\sigma') \cdot \mathsf{c}'(\#\mathsf{tick}(\sigma'_{\mathsf{s}}, \gamma(\sigma'_{\mathsf{s}}))) \\
&= \begin{cases} \mathcal{E}(\sigma') & \text{if } \gamma(\sigma'_{\mathsf{s}}) = \mathsf{skip} \\ \mathcal{E}(\sigma' \cdot (\#\mathsf{tick}(\sigma'_{\mathsf{s}}), D)) & \text{if } \gamma(\sigma'_{\mathsf{s}}) = D \end{cases}
\end{aligned}$$

  Similarly to the reactive case above, the final step follows from how the proactive update function γ is defined by the enforcer update function μ and how μ is used by the enforcement algorithm to issue PCom and NoCom commands.

Finally, the finite case shows that there cannot exist an infinite system trace for which Lemma 3.27 does not hold. For any infinite trace $\sigma_{\mathsf{s}} \in \mathsf{traces}_{\omega}(\Xi(\mathcal{E}), \varepsilon_{\mathsf{s}})$ we have shown that for all finite prefixes $\sigma_{\mathsf{s}|\ldots i}$ with an arbitrary $i \in \mathbb{N}$, there exists an MFOTL trace $\sigma_f \in \mathbb{T}_f$ such that $\mathsf{c}(\sigma_{\mathsf{s}|\ldots i}) = \mathcal{E}(\sigma_f)$. If, for some infinite trace $\sigma_{\mathsf{s}} \in \mathsf{traces}_{\omega}(\Xi(\mathcal{E}), \varepsilon_{\mathsf{s}})$, there would not exist a trace

$\sigma_\omega \in \mathbb{T}_\omega$ such that $\mathsf{c}(\sigma_\mathbf{s}) = \mathcal{E}(\sigma_\omega)$, then there would exist some finite prefix $\sigma_{\mathbf{s}|\ldots i} \in \mathsf{traces}_f(\Xi(\mathcal{E}), \varepsilon_\mathbf{s})$ such that there also does not exist a trace $\sigma_f \in \mathbb{T}_f$ such that $\mathsf{c}(\sigma_{\mathbf{s}|\ldots i}) = \mathcal{E}(\sigma_f)$. $\square$

Use of Generative AI Tools

AI Tool	Use Case	Scope	Remarks
GitHub Copilot	Code completion, Code suggestions, Error explanations, Exploring code base	Lex code base: OCaml, JavaScript Report: LaTeX (error explanations)	Various models: GPT-4.1, GPT-4o, o3-mini, Claude 3.5 Sonnet, Gemini 2.0 Flash, Gemini 2.0 Pro
DeepL Write	Correction of grammatical mistakes, styling	Entire report	Free Trial
OpenAI ChatGPT	Brainstorming, ”interactive” web search	Introduction	”Deep Research” mode

Figure 5: Overview of AI tools used in this thesis