Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Monitoring Complex Data Types

Bachelor Thesis

Remo Zumsteg

September 07, 2022

Professor: Prof. Dr. David Basin
Advisors: Dr. Srđan Krstić, François Hublet and Joshua Schneider
Department of Computer Science, ETH Zürich

Abstract

The runtime verification tool MonPoly with its extensive temporal specification language MFODL provides a broad set of features, including real-time monitoring, global quantification, regular expressions matching over program traces, and computing SQL-like aggregations. While JSON-formatted application logs have become ubiquitous, allowing systems to output events as arbitrary complex data structures, both MonPoly and MFODL lack support for custom event data types.

To rectify this, we present an extension of MFODL that allows users to formulate policies over events of custom product types. A type-checking algorithm accompanies the extension with full type-inference support. Moreover, we provide an extension of MonPoly that supports monitoring JSON-formatted application logs with no or minimal preprocessing. We accomplish this with a formula compiler and a generalized log stream preprocessor.

Our case study on two real-world scenarios shows that our extension not only simplifies the monitoring process of complex-typed event streams but it allows users to formulate specifications over complex data structures more naturally, increasing the readability and comprehensibility of monitoring policies.

Acknowledgment

I would like to thank my advisors, Srđan Krstić and François Hublet, for making my thesis project possible in the first place and for supporting me endlessly during the last six months. You have been strongly invested in my project, and I felt supported at all times. I am very grateful for your time and your patience. I would also like to extend my deepest gratitude to Joshua Schneider: your profound insight into MonPoly and MFODL was tremendously helpful. Time and again, you steered the project in the right direction and helped me appreciate the purpose of my work.

Finally, I thank the professors at ETH Zurich, especially Prof. David Basin, Prof. Peter Müller, Prof. Zhendong Su, and Prof. Ueli Maurer. Your lectures were pure joy and taught me all the fundamentals I needed for my thesis.

Contents

Chapter 1

Introduction

Validating software applications and systems for correct behavior can be done in many ways. Automated or manual testing, formal verification, and model checking are well-known techniques that ensure the correctness of software and are usually part of the development process. Runtime verification is a lightweight formal verification technique mainly concerned with checking the behavior of a system during its runtime. Runtime verification generally validate that a given program trace adheres to predefined properties.

In the heart of runtime verification lies the *monitor*. It accepts one or many properties formulated in a specification language, a program trace to monitor, and returns a verdict indicating if the given trace satisfies the formulated properties. The process of gathering a program trace may differ based on the architecture of an application: if runtime verification is part of the monitored system itself, a stream of internal events can be monitored directly. In other cases, application logs may approximate the actual program trace.

Similarly, there is a variety of specification languages for defining the set of desired and undesired program traces. While we focus on temporal logic-based languages, others may be based on regular expressions, streams, or a combination of these families [3]. Early iterations of temporal logic-based specification languages, such as CARET [1], regard events of a program trace as atomic symbols, for instance, openFile or closeFile. Later, languages were established that support the formalization of properties on streams of events that carry additional data [17].

Depending on the syntax and semantics of the specification language and the capabilities of the underlying monitor, the domain of event data may be limited. While some runtime verification tools such as HLola [12], ParTraP [9], and LogScope [2] provide support for custom event data types, many available tools limit the domain to primitive types, namely boolean, numerical, and string types. In contrast, JSON formatted logs have become ubiquitous,

allowing applications to output events as arbitrarily complex data structures. To monitor complex typed event streams using a runtime verification tool with limited type support, the event stream must first be transformed into a supported format. We distinguish two possible approaches: First, a general preprocessor transforms the event stream, independent of the application-specific semantics of the data structures carried by events. The preprocessor can thus be reused, but the transformation increases the complexity of the expressed properties in the given specification language. Second, an application- and property-specific preprocessor extracts only relevant event data into a supported format. In this case, the formalization of properties in the specification language is generally less complex, but the preprocessor must be potentially adjusted or even rewritten whenever the desired properties or application log formats change. Both approaches increase the risk of introducing subtle bugs into the monitoring process.

To improve the runtime verification of complex typed event streams, we extend the existing monitor system MonPoly [7] and its specification language metric first-order dynamic logic (MFODL) [4]. The goal is to support monitoring MFODL properties over JSON-formatted application logs as an event source, without or only minimal preprocessing. At the same time, specifications should be expressible naturally, hence improving the development experience and readability of formulas. We achieve this by the following contributions:

- Language extension: We introduce complex-typed MFODL (CMFODL) by extending the syntax and semantics of MFODL. CMFODL supports variables and constants of custom product types. Projections on variables can be used to access values of nested fields.
- CMFODL compiler: We implement a formula compiler, translating CMFODL properties to semantically equivalent MFODL formulas, which can be consumed and monitored by MonPoly.
- JSON log parser: A dedicated log parser for JSON logs that acts as a generalized event stream preprocessor: The stream of complex-typed event data is transformed into a stream of finite relations, as expected by MonPoly and the compiler.

Compared to existing temporal specification languages with support for custom typed event data, CMFODL inherits MFODL features such as global quantification, referencing past event data and support for real-time monitoring, providing a unique combination of features.

Writing formulas is hard, especially those that belong to a monitorable fragment supported by MonPoly. We improve the usability of CMFODL with the following contributions:

- A static type-checking algorithm with type inference support, helping

users in writing correct formulas while keeping the additional effort minimal.
- An extended monitorability check that helps users to better understand why a given CMFODL formula is not monitorable by MonPoly.

The structure of this thesis is as follows: Chapters 11 and 2 present related work and give an overview of the background to this work. Chapters 3 and 4 describe the language extension CMFODL and formalize its type system. Chapters 5 and 6 introduce the preprocessing of event streams and compilation of CMFODL formulas. Chapter 7 gives an overview of the changes introduced to the monitoring tool MonPoly. Finally, Chapter 8 presents a case study on existing approaches of generalized and specialized preprocessing, and demonstrates the added value of this work in real-world scenarios.

Chapter 2

Background

This chapter provides the background knowledge helpful for comprehending the following chapters. We first give an overview of runtime verification in general and afterward describe the specification language MFODL and the runtime monitoring system MonPoly in more detail.

2.1 Runtime verification

Runtime verification (RV) is pursued as a complementary verification strategy besides other approaches such as testing or formal verification. Compared to other strategies, runtime verification can be used to verify the correct behavior of a system during its runtime [20]. In general, we want to detect deviations in the behavior of a system from its specification. On the detection of divergence, a runtime verification tool may solely act as a reporter or influence the run of the system under monitoring, where the latter is known as enforcement [20]. Formally, a run of a system can be modeled as a possibly infinite sequence of program states [20], from now on called program trace. A *correctness property* is equal to a subset of program traces that adhere to a given system specification.

A *monitor* is an algorithm that, in its simplest form, produces a boolean verdict, indicating if the current program trace of a monitored system is an element of a given correctness property. We further differentiate between online and offline monitoring: In the former, the monitor checks the execution of a system in real time by incrementally processing new system states from an ongoing program trace. In the latter, a monitor analyses a (finite) program trace of a previous system run. During online monitoring of an infinite trace, some properties may not always be verifiable in finite time. For example, if a *liveness property* is not satisfied by an infinite trace, there exists no finite prefix of the trace acting as a witness for refutation [16].

Usually, a monitor is automatically generated based on a higher-level specification [20]. When we use the term *specification language*, we imply a language to formulate the specification for generating a monitor. MFODL and the extension CMFODL, presented in this work, are specification languages. In contrast, MonPoly is an implementation of a monitor generator from MFODL specifications [7]. From now on, we call specifications written in MFODL *formulas* and MonPoly the *monitoring system*.

2.2 MFODL

MFODL [4] is the result of combining *Metric Dynamic Logic* (MDL) [11] and *Metric First-Order Temporal Logic* (MFOTL) [10] with additional features [23]. While it inherits the temporal operators from MFOTL, it also supports regular expression over program traces introduced by MDL. Further features include aggregation operators introduced to MFOTL by Basin et al. [6].

MFODL formulas are interpreted over a temporal structure consisting of a sequence of tuples, each consisting of a timestamp and a finite set of relations. Values of relations are elements of a possibly infinite domain $|\mathcal{D}|$. The formula $r(a_1, a_2, \ldots, a_k)$ for a k-ary predicate r is evaluated as true at a time point i for values $a_1, a_2, \ldots, a_k \in |\mathcal{D}|$, whenever the tuple $(a_1, a_2, \ldots, a_k)$ is an element of the relation r at the i-th element of the temporal structure. While the domain $|\mathcal{D}|$ is not fixed, one important limitation of MFODL is the fact that each value in $|\mathcal{D}|$ is regarded to be atomic: neither the syntax nor the semantics of MFODL allows accessing fields of compound values. The extension CMFODL presented in this work improves upon this. Section 3.1 provides an overview of the supported syntax, and Section 3.2 formalizes the semantics of CMFODL formulas.

2.3 MonPoly

MonPoly is a runtime verification tool accepting specifications in MFODL. To effectively monitor policies, MonPoly only handles a syntactic fragment of MFODL [7]. For once, all intervals of future temporal operators must be bounded. Furthermore, all intermediate results must be finite when evaluating a formula bottom-up. Finally, the domain of values accepted as part of a relation in a temporal structure is limited to the primitive types `string`, `int`, `float`, and `regexp` [7].

Besides a program trace, MonPoly requires a signature file and a formula file as input. The signature file describes the first-order signature used by the formula. The formula file contains the specification written in MFODL [7]. The accepted grammar of these files is described in Sections 7.1.1 and 7.1.2. The log to monitor can either be provided as a file for offline monitoring or as

a stream to standard input for online monitoring. The log represents a finite temporal structure by enumerating the relations tuple by tuple for each time point in increasing order. This work extends MonPoly such that it accepts specifications written in CMFODL and can parse JSON formatted log files.

Chapter 3

Complex-Typed MFODL

As outlined in the introduction, MonPoly makes use of MOFDL as its policy specification language [7]. To support monitoring complex-structured application logs using MFODL policies, we propose the extension CMFODL of MFODL with support for complex data types. Section 3.1 describes the abstract syntax of CMFODL, whereas Section 3.2 presents the semantics of the newly introduced syntax.

3.1 Abstract syntax

We first introduce the syntax of complex-typed metric first-order dynamic logic (CMFODL). Here we diverge from the commonly used, minimalistic single-sorted definition [5] and opt to cover additional details as specified in the monitoring tool MonPoly [7].

A sorted first-order signature defines constant, function, and predicate symbols by enumerating their names and respective sorts. Intuitively, a sort describes a subset of domain values that will be used to interpret the symbols in the semantics. The signature assigns a single sort to each constant symbol. Predicate and function symbols require a sequence of sorts that characterize their arguments; function symbols have an additional sort for their result. Here we opt not to include function symbols in the signature. We incorporate them into the syntax and define their sorts indirectly using a type system later.

We fix an infinite set of names N and a set $\mathsf{PS} = \{\mathsf{Z}, \mathsf{F}, \mathsf{Str}, \mathsf{RE}, \mathsf{B}, \mathsf{Null}\}$ of six primitive sorts: integers (Z), floats (F), strings (Str), regular expressions (RE) booleans (B) and the singleton sort Null consisting of the single constant value null. We assume that N and PS are disjoint. A sorted first-order signature Δ is a tuple $(S, S_{\text{def}}, C, P)$, where S is a finite subset of $\mathsf{N} \cup \mathsf{PS}$, presenting the sorts used by the signature. The partial mapping S_{def} defines the non-

primitive sorts; we will explain it below. The partial mapping $C \in \mathsf{N} \rightharpoonup S$ assigns a sort to each constant symbol. The available constant symbols are implicitly given by $\mathsf{dom}(C)$, the domain of C. Finally, $P \in \mathsf{N} \rightharpoonup \mathsf{PS}^*$ assigns a list of (primitive) sorts to each predicate symbol in $\mathsf{dom}(P)$. For example, $\Delta = (\mathsf{PS}, \emptyset, \{\mathsf{r} \mapsto \mathsf{Z}\}, \{\mathsf{publish} \mapsto (\mathsf{Z}), \mathsf{approve} \mapsto (\mathsf{Z})\})$ defines the signature for a simple reporting system, where relevant predicate symbols correspond to publishing and approving reports with integer IDs, and we have an integer constant r. There is no custom sort; hence we specified the empty mapping $\emptyset$ for S_{def}.

We allow the definition of custom-named sorts in the signature. In particular, named products of sorts with named fields can be defined as a custom sort. The set of names N can be used for custom sort names and their field names. The structure of a custom sort is described by a *sort schema*. The set Sch of all sort schemas is the smallest set such that if f is a function from a finite (possibly empty) subset of N to $\mathsf{N} \cup \mathsf{PS}$, then $\Pi f \in \mathsf{S}$. Note that Π is just a symbol that marks the schema as a product. We usually write product schemas by enumerating the pairs of field names and their sorts, e.g. $\{n_1 : s_1, n_2 : s_2, \ldots, n_k : s_k\}$ is a product schema with k fields. If s is a sort schema, we write $\mathsf{ran}(s)$ for the set of sort names occurring in s. For example, $\mathsf{ran}(\Pi f) = \mathsf{ran}(f)$, the range of f. The sort definition S_{def} is now a partial mapping $\mathsf{CS} \rightharpoonup \mathsf{Sch}$ with $\mathsf{CS} = S - \mathsf{PS}$, subject to a well-formedness constraint. It ensures that all sorts used in sort definitions are defined and that there is no recursion between sorts. More precisely, there must be a sequence $S_1 \subset S_2 \subset \cdots \subset S_k$ such that

- $S_1 \subseteq \mathsf{PS}$,
- $S_k = \mathsf{dom}(S_{\text{def}})$, and
- $S_{i+1} \setminus S_i = \{s_i\}$ for all $i < k$ and some s_i, where $\mathsf{ran}(S_{\text{def}}(s_i)) \subseteq S_i$.

Since we allow custom products of sorts, one can easily encode predicate symbols using this more general mechanism. We therefore simplify the signature to the triple (S, S_{def}, C). For the reporting system example above, the new signature is $(\mathsf{PS} \cup \{\mathsf{publish}, \mathsf{approve}\}, \{\mathsf{publish} \mapsto \{\_1 : \mathsf{Z}\}, \mathsf{approve} \mapsto \{\_1 : \mathsf{Z}\}\}, \{\mathsf{r} \mapsto \mathsf{Z}\})$. It has two custom sorts, one for each predicate symbol, that are defined as products with a single field _1. In the following, we assume that N contains a name of the form $\_k$ for every natural number $k \geq 1$. Thus we can perform the encoding for predicates of any arity. A sort s such that $S_{\text{def}}(s)$ is a product (exclusively) over fields _1 to $\_k$ is called a *tuple sort*. The particular case of an empty product is also a tuple sort. We define the arity $\iota(s)$ of a tuple sort s to be n and leave it undefined for all other sorts.

To define the syntax of formulas, we further fix a countably infinite set V of variables and the set I of nonempty intervals $[a,b) := \{x \in \mathbb{N} \mid a \leq x < b\}$, where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, and $a < b$. Given a signature (S, S_{def}, C), formulas φ are defined inductively, where s, c, v, n, $\bar{v}$, $n\bar{v}$, and I range over S, $\mathsf{dom}(C)$, V, N, V^*,$(\mathsf{N} \times \mathsf{V})^*$, and I, respectively. Figure 3.1 formalizes syntax of MFODL,

$$
\begin{array}{rcl}
t & ::= & c \mid v \mid \mathsf{f2i}(t) \mid \mathsf{i2f}(t) \mid \mathsf{r2s}(t) \mid \mathsf{s2r}(t) \mid \mathsf{i2s}(t) \mid \mathsf{s2i}(t) \mid \mathsf{f2s}(t) \mid \mathsf{s2f}(t) \\
 & & -t \mid t+t \mid t-t \mid t*t \mid t/t \mid t\%t \mid t.n \mid s\overbrace{\{n:t,\ldots,n:t\}}^{=S_{\text{def}}(s)} \\
\varphi & ::= & s(t) \mid \downarrow s(t) \mid \\
 & & \bot \mid \top \mid s(\overbrace{t,\ldots,t}^{\iota(s)}) \mid t \approx t \mid t \prec t \mid t \preceq t \mid t \overset{\mathsf{S}}{\Leftarrow} t \mid t \overset{\mathsf{RE}}{\Leftarrow} t(t,\ldots,t) \mid \\
 & & \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \mid \exists \bar{v}.\ \varphi \mid \forall \bar{v}.\ \varphi \mid \\
 & & \bullet_I \varphi \mid \circ_I \varphi \mid \blacklozenge_I \varphi \mid \lozenge_I \varphi \mid \blacksquare_I \varphi \mid \square_I \varphi \mid \varphi\, \mathsf{S}_I\, \varphi \mid \varphi\, \mathsf{U}_I\, \varphi \mid \blacktriangleleft_I r \mid \triangleright_I r \mid \\
 & & v \leftarrow \Omega\ v; \bar{v}\ \varphi \mid \mathsf{let}\ s\overbrace{\{n:v,\ldots,n:v\}}^{=S_{\text{def}}(s)} = \varphi\ \mathsf{in}\ \varphi \mid \mathsf{let}\ n(\bar{v}) = \varphi\ \mathsf{in}\ \varphi \\
r & ::= & \cdot \mid \varphi? \mid r\ r \mid r+r \mid r^*
\end{array}
$$

Figure 3.1: MFODL syntax definition

where $\Omega \in \{\mathsf{CNT}, \mathsf{SUM}, \mathsf{AVG}, \mathsf{MED}, \mathsf{MIN}, \mathsf{MAX}\}$ stands for an aggregation operator. t relates to terms, φ to formulas, and r to regular expressions over program traces. Sort s in a subformula of the form $\mathsf{let}\ s\{n:t,\ldots,n:t\} = \varphi\ \mathsf{in}\ \psi$ or $\mathsf{let}\ n(\bar{v}) = \varphi\ \mathsf{in}\ \varphi$ cannot occur in φ or outside of the subformula. Let $\mathsf{fv}(\varphi)$ denote a set of free variables of an MFODL formula φ, defined as usual, whereas the set Terms represents the set of all MFODL terms.

3.2 Semantics

MFODL formulas are interpreted over *temporal structures (TS)*, which model timestamped and totally ordered sequences of observations.

Given a signature Δ, each observation in TS consists of a timestamp and a finite first-order Δ-structure $\mathcal{D}$, which interprets (i.e., associates appropriate values to) the elements of the signature. In $\mathcal{D}$ each sort $s \in S$ is interpreted with a nonempty domain d of values each tagged with its sort, i.e., $s^{\mathcal{D}} = d$ ‘ $\lambda e.\ (s,e)$. In particular, primitive sorts have the expected domains: the sort of integers is interpreted as the set of tagged integers ($\mathsf{Z}^{\mathcal{D}} = \mathbb{Z}$). Similar holds for floats ($\mathsf{F}^{\mathcal{D}} = \mathbb{F}$), strings ($\mathsf{Str}^{\mathcal{D}} = \mathbb{S}$), and regular expressions ($\mathsf{RE}^{\mathcal{D}} = \mathbb{RE}$). The interpretation function $\cdot^{\mathcal{D}}$ is lifted for set operations like unions and products. To define the interpretation of a custom product sort $s \in S - \mathsf{PS}$, we define $S_{\text{def}}(s) = \Pi f$ as $\left[\prod_{n \in \text{dom}(f)} f(n)^{\mathcal{D}}\right]$ ‘ $\lambda e.\ (s,e)$.

Given a signature $\Delta = (S, S_{def}, C)$, the finite first-order $\Delta-$structure is the triple $\mathcal{D} = (\mathbb{D}, \mathbb{C}, \mathbb{O})$ where $\mathbb{D} = \bigcup_{s \in S} s^{\mathcal{D}}$ is union of values of all domains. For every $(c,s) \in C$ we have $c^{\mathcal{D}} \in s^{\mathcal{D}}$ in $\mathbb{C}$ and we have a finite set of objects $\mathbb{O} \subseteq \mathbb{D}$. We denote a closure of $\mathbb{O}$ with respect to the projection operator (.) as the smallest set $\mathbb{O}\downarrow$ such that:

- $\mathbb{O} \subseteq \mathbb{O}\downarrow$
- if $(e,f) \in \mathbb{O}\downarrow$ then $o \in \mathbb{O}\downarrow$ for all $(\_,o) \in f$

A temporal structure ρ is then an infinite sequence (or a stream) $(\tau_i, \mathcal{D}_i)_{i\in\mathbb{N}}$ of finite first-order Δ-structures $\mathcal{D}_i$ with associated time-stamps τ_i. We refer to a component of a structure at a specific time-point i using a subscript (e.g., $\mathbb{D}_i$). All finite first-order Δ-structures $\mathcal{D}_i$ agree on their sorts ($\forall i.\ \forall s \in S.\ s^{\mathcal{D}_i} = s^{\mathcal{D}_{i+1}}$) and constant interpretations ($\forall i.\ \mathbb{C}_i = \mathbb{C}_{i+1}$). Since they agree on their sorts, they also agree on their domains ($\forall i.\ \mathbb{D}_i = \mathbb{D}_{i+1}$), When referring to these common components in a TS, we omit the reference to a particular time-point (e.g., we write just $\mathbb{D}$).

Time-stamps are discrete, modeled as natural numbers $\tau_i \in \mathbb{N}$. We allow the event source to use finitely precise clocks: structures at different time-points $i \neq j$ may have the same time-stamp $\tau_i = \tau_j$. The sequence of time-stamps must be non-strictly increasing ($\forall i.\ \tau_i \leq \tau_{i+1}$) and always eventually strictly increasing ($\forall \tau.\ \exists i.\ \tau < \tau_i$).

The valuation v is a mapping $\mathsf{V} \to \mathbb{D}$, assigning domain elements to variables. Overloading notation, v can be applied to terms:

- $v(c) \in \mathbb{D}$;
- $v(\mathsf{f2i}(t)) = \mathsf{int\_of\_float}(v(t))$; $v(\mathsf{i2f}(t)) = \mathsf{float\_of\_int}(v(t))$;
- $v(\mathsf{r2s}(t)) = \mathsf{str\_of\_regex}(v(t))$; $v(\mathsf{s2r}(t)) = \mathsf{regex\_of\_str}(v(t))$;
- $v(\mathsf{i2s}(t)) = \mathsf{int\_to\_str}(v(t))$; $v(\mathsf{s2i}(t)) = \mathsf{str\_to\_int}(v(t))$;
- $v(\mathsf{f2s}(t)) = \mathsf{float\_to\_str}(v(t))$; $v(\mathsf{s2f}(t)) = \mathsf{str\_to\_float}(v(t))$;
- $v(-t) = -v(t)$;
- $v(t \bowtie t') = v(t) \bowtie v(t')$ for $\bowtie \in \{+,-,*,/,\%\}$;
- $v(t.f) = d$ if $v(t) = (n,o)$ and $(f,d) \in o$.
- $v(s\{n_1 : t_1, \ldots, n_k : t_k\}) = (s, \{(n_i, v(t_i)) \mid i = 1,\ldots,k\})$

We write $v[x \mapsto y]$ for the function equal to v, except that the argument x is mapped to y. For a vector of free variables $\bar{u} = [u_1,\ldots,u_k]$, we write $v(\bar{u})$ for the tuple $(v(u_1),\ldots v(u_k))$. We also define a let mapping δ as a partial function $\delta : \mathsf{N} \rightharpoonup (\mathbb{N} \to 2^{\mathbb{D}})$.

Figure 3.2 describes the semantics of the operators defined in the MFODL syntax. The relation $\delta, v, i \models_\rho \varphi$ defines the satisfaction of the formula φ for a let mapping δ, valuation v, and a time-point i with respect to the temporal structure ρ. Whenever ρ is fixed and clear from the context, we omit the subscript on $\models$. We focus only on the core subset of the operators, while the rest of the operators are shorthands. Temporal operators with no interval have $[0,\infty)$ instead.

$\delta, v, i \models s(t)$	iff $v(t) = (s, \_)$ and if $\delta(\mathsf{s})(i) \neq \bot$ then $v(t) \in \delta(\mathsf{s})(i)$ else $v(t) \in \mathbb{O}^{\mathcal{D}_i}$
$\delta, v, i \models \downarrow \mathsf{s}(t)$	iff $v(t) = (s, \_)$ and if $\delta(\mathsf{s})(i) \neq \bot$ then $v(t) \in \delta(\mathsf{s})(i)\downarrow$ else $v(t) \in \mathbb{O}^{\mathcal{D}_i}\downarrow$
$\delta, v, i \models t \approx t'$	iff $v(t) = v(t')$
$\delta, v, i \models t \preceq t'$	iff $v(t) \leq v(t')$
$\delta, v, i \models t \overset{\mathsf{RE}}{\Leftarrow} t'(t_1, \ldots, t_n)$	iff $v(t)$ matches $v(t')$with capture group i valued as $v(t_i)$
$\delta, v, i \models \neg\varphi$	iff $\delta, v, i \not\models \varphi$
$\delta, v, i \models \varphi \wedge \psi$	iff $\delta, v, i \models \varphi$ and $\delta, v, i \models \psi$
$\delta, v, i \models \exists x.\, \varphi$	iff $\delta, v[x \mapsto z], i \models \varphi$ for some $z \in \mathbb{D}$
$\delta, v, i \models \bullet_I \varphi$	iff $i > 0$, $\tau_i - \tau_{i-1} \in I$, and $\delta, v, i-1 \models \varphi$
$\delta, v, i \models \circ_I \varphi$	iff $\tau_{i+1} - \tau_i \in I$ and $\delta, v, i+1 \models \varphi$
$\delta, v, i \models \varphi \,\mathsf{S}_I\, \psi$	iff $\delta, v, j \models \psi$ for some $j \leq i$, $\tau_i - \tau_j \in I$, and $\delta, v, k \models \varphi$ for all k with $j < k \leq i$
$\delta, v, i \models \varphi \,\mathsf{U}_I\, \psi$	iff $\delta, v, j \models \psi$ for some $j \geq i$, $\tau_j - \tau_i \in I$, and $\delta, v, k \models \varphi$ for all k with $i \leq k < j$
$\delta, v, i \models \mathsf{let}\ s\{\bar{nu}\} = \varphi\ \mathsf{in}\ \psi$	iff $\delta[s \to f], v, i \models \psi$ where $f = \lambda i.\{(s, \{n_1 : w(u_1), \ldots, n_k : w(u_k)\}) \mid \delta, w, i \models \varphi\}$
$\delta, v, i \models \blacktriangleleft_I r$	iff $\tau_j - \tau_i \in I$ and $(i, j) \in \mathcal{R}(r)$ for some $j \geq i$
$\delta, v, i \models \triangleright_I r$	iff $\tau_i - \tau_j \in I$ and $(j, i) \in \mathcal{R}(r)$ for some $j \leq i$

$$
\begin{aligned}
\mathcal{R}(\star) &= \{(i, i+1) \mid i \in \mathbb{N}\} \\
\mathcal{R}(\varphi?) &= \{(i, i) \mid \delta,\ v,\ i \models \varphi\} \\
\mathcal{R}(r + s) &= \mathcal{R}(r) \cup \mathcal{R}(s) \\
\mathcal{R}(r \cdot s) &= \{(i, k) \mid \exists j.\ (i, j) \in \mathcal{R}(r) \text{ and } (j, k) \in \mathcal{R}(s)\} \\
\mathcal{R}(r*) &= \{(i, i) \mid i \in \mathbb{N}\} \cup \left\{(i_0, i_k) \mid \exists i_1, \ldots, i_k.\ (i_j, i_{j+1}) \in \mathcal{R}(r) \text{ for all } 0 \leq j < k\right\}
\end{aligned}
$$

$x \prec y := x \preceq y \wedge \neg x \approx y$, $\bot := 0 \approx 1$, $\top := \neg\bot$, $\mathsf{s}(t_1, \ldots, t_n) = \mathsf{s}(\mathsf{s}\{\_1 : t_1, \ldots, \_\mathsf{n} : t_n\})$, $t \overset{\mathsf{S}}{\Leftarrow} t' := t \overset{\mathsf{RE}}{\Leftarrow} t'()$, $\varphi \vee \psi := \neg\varphi \wedge \neg\psi$, $\varphi \to \psi := \neg\varphi \vee \psi$, $\varphi \leftrightarrow \psi := \varphi \to \psi \wedge \psi \to \varphi$, $\forall x.\, \varphi := \neg\exists x.\, \neg\varphi$, $\blacklozenge_I \varphi := \top \,\mathsf{S}_I\, \varphi$, $\Diamond_I \varphi := \top \,\mathsf{U}_I\, \varphi$, $\Box_I \varphi := \neg\Diamond_I \neg\varphi$, $\blacksquare_I \varphi := \neg\blacklozenge_I \neg\varphi$, and $\mathsf{let}\ \mathsf{p}(t_1, \ldots, t_n) = \varphi\ \mathsf{in}\ \psi := \mathsf{let}\ \mathsf{p}(\{\_1 : t_1, \ldots, \_\mathsf{n} : t_n\}) = \varphi\ \mathsf{in}\ \psi$.

Figure 3.2: MFODL operator semantics

Chapter 4

Type Checking

This chapter specifies the concept of well-formed CMFODL formulas and explains in more detail how type checking and type inference are implemented in MonPoly.

4.1 Well-typed terms and well-formed formulas

Notice that the current definition of the CMFODL syntax described in Section 3.1 allows for writing formulas that do not have well-defined semantics. For example, $x \approx y \wedge (x \prec 3) \wedge (y \approx \text{"}two\text{"})$. Ideally, we would like to recognize such formulas as malformed and reject them.

Well-formed CMFODL formulas have well-typed terms. Here type and sort are synonymous. We use the following type-language for the CMFODL terms:

$$\begin{array}{rcl} T & ::= & \mathsf{Z} \mid \mathsf{F} \mid \mathsf{Str} \mid \mathsf{RE} \mid \mathsf{B} \mid \mathsf{Null} \mid \mathsf{N}\ \{\ FS\ \} \mid \{\ FS\ \} \\ FS & ::= & \mathsf{N} : T \mid \mathsf{N} : T,\ FS \end{array}$$

The case $\mathsf{N}\{\ FS\ \}$ corresponds to named product types, while $\{\ FS\ \}$ corresponds to unnamed product types. We further establish a partial order over unnamed product types as $f_1 \leq f_2$ iff $f_2 \subseteq f_1$ and between named and unnamed product types as $(e, f) \leq f$.

The set $\mathsf{Num} = \{\mathsf{Z}, \mathsf{F}\}$ is the numeric type class, $\mathsf{Ord} = \{\mathsf{Z}, \mathsf{F}, \mathsf{Str}, \mathsf{RE}\}$ the type class of all totally-ordered types, while Any is the type class of all types. Furthermore, let Γ be the symbol table, i.e., the set of bindings $x_i : \tau_i$ representing a partial map from variables to types. Given a symbol table Γ, an CMFODL term t and a type τ, type judgement $\Gamma \vdash t :: \tau$ means that t is well-typed in context of Γ and has type τ. Figure 4.1 lists all type inference rules for CMFODL terms.

$$\frac{}{\Gamma, x : \tau \vdash x :: \tau}\ \textsc{Var} \qquad \frac{c \in \mathsf{Z}}{\Gamma \vdash c :: \mathsf{Z}}\ \textsc{Int} \qquad \frac{c \in \mathsf{F}}{\Gamma \vdash c :: \mathsf{F}}\ \textsc{Flt}$$

$$\frac{c \in \mathsf{RE}}{\Gamma \vdash c :: \mathsf{RE}}\ \textsc{Regex} \qquad \frac{c = \mathsf{True}}{\Gamma \vdash c :: \mathsf{B}}\ \textsc{True} \qquad \frac{c = \mathsf{False}}{\Gamma \vdash c :: \mathsf{B}}\ \textsc{False}$$

$$\frac{c = \mathsf{null}}{\Gamma \vdash c :: \mathsf{Null}}\ \textsc{Null} \qquad \frac{\Gamma \vdash t :: \tau \qquad \tau \in \mathsf{Num}}{\Gamma \vdash -t :: \tau}\ \textsc{UnOp} \qquad \frac{\Gamma \vdash t :: \tau' \qquad \tau' \leq \{n : \tau\}}{\Gamma \vdash t.n :: \tau}\ \textsc{Proj}$$

$$\frac{\Gamma \vdash t :: \mathsf{F}}{\Gamma \vdash \mathsf{f2i}(t) :: \mathsf{Z}}\ \textsc{f2i} \qquad \frac{\Gamma \vdash t :: \mathsf{Z}}{\Gamma \vdash \mathsf{i2f}(t) :: \mathsf{F}}\ \textsc{i2f} \qquad \frac{\Gamma \vdash t :: \mathsf{F}}{\Gamma \vdash \mathsf{f2s}(t) :: \mathsf{Str}}\ \textsc{f2s}$$

$$\frac{\Gamma \vdash t :: \mathsf{Str}}{\Gamma \vdash \mathsf{s2f}(t) :: \mathsf{F}}\ \textsc{s2f} \qquad \frac{\Gamma \vdash t :: \mathsf{Z}}{\Gamma \vdash \mathsf{i2s}(t) :: \mathsf{Str}}\ \textsc{i2s} \qquad \frac{\Gamma \vdash t :: \mathsf{Str}}{\Gamma \vdash \mathsf{s2i}(t) :: \mathsf{Z}}\ \textsc{s2i}$$

$$\frac{\Gamma \vdash t :: \mathsf{RE}}{\Gamma \vdash \mathsf{r2s}(t) :: \mathsf{Str}}\ \textsc{r2s} \qquad \frac{\Gamma \vdash t :: \mathsf{Str}}{\Gamma \vdash \mathsf{s2r}(t) :: \mathsf{RE}}\ \textsc{s2r}$$

$$\frac{\Gamma \vdash t :: \tau \qquad \Gamma \vdash t' :: \tau \qquad \tau \in \mathsf{Num}}{\Gamma \vdash t \oplus t' :: \tau}\ \textsc{BinOp} \quad \oplus \in \{+, -, *, /, \%\}$$

Figure 4.1: Typing rules for MFODL terms

Similarly, given a predicate schema Δ and a formula φ (or temporal regular expression r), judgement $\Delta;\ \Gamma \vdash \varphi$ (or $\Delta;\ \Gamma \vdash r$) means that φ (or r) is a well-formed formula (or temporal regular expression) in context of Δ and Γ. Given fresh (i.e. not in Γ) types $\tau, \tau_1, \ldots, \tau_n \in \mathsf{Any}$, and a list of fresh types $\bar{\sigma} \in \mathsf{Any}^*$, typing rules for CMFODL are listed in Figure 4.2. Expression $\bar{zs} : \bar{\sigma}$ is a shorthand for $\{(\bar{zs}[i] : \bar{\sigma}[i]) \mid i \in \{1, \ldots, |\bar{zs}|\}\}$ assuming $|\bar{zs}| = |\bar{\sigma}|$.

When we write $\Gamma,\ x : t$, it means that we pattern match on the current symbol table to assert that it contains the binding $x : t$. If we similarly use the signature, we either assert the existence of a predicate symbol (e.g., as in the Pred rule) or the existence of a custom sort (e.g., as in the Sort rule).

$$\frac{\Gamma \vdash t :: s\{f\}}{\Delta, s\{f\};\ \Gamma \vdash s(t)}\ \textsc{Sort} \qquad \frac{\Gamma \vdash t :: \mathsf{Str} \qquad \Gamma \vdash t' :: \mathsf{Str}}{\Delta;\ \Gamma \vdash t \overset{\mathsf{S}}{\Leftarrow} t'}\ \textsc{Substring}$$

$$\frac{\Gamma \vdash t :: \tau \qquad \Gamma \vdash t' :: \tau}{\Delta;\ \Gamma \vdash t = t'}\ \textsc{Equal} \qquad \frac{\Gamma \vdash t :: \tau \qquad \Gamma \vdash t' :: \tau \qquad \tau \in \mathsf{Ord}}{\Delta;\ \Gamma \vdash t \bowtie t'}\ \textsc{OrdRel} \quad \bowtie \in \{<, \leq\}$$

$$\frac{\Gamma \vdash t :: \mathsf{Str} \qquad \Gamma \vdash t' :: \mathsf{RE} \qquad \forall i.\ \Gamma \vdash t_i :: \mathsf{Str}}{\Delta;\ \Gamma \vdash t \overset{\mathsf{RE}}{\Leftarrow} t'\ (t_1, \ldots, t_n)}\ \textsc{Match} \qquad \frac{\forall i.\ \Gamma \vdash t_i :: \tau_i}{\Delta, (p, (\tau_1, \ldots, \tau_n));\ \Gamma \vdash p(t_1, \ldots, t_n)}\ \textsc{Pred}$$

$$\frac{\Delta;\ \Gamma \vdash \varphi}{\Delta;\ \Gamma \vdash \star\varphi}\ \textsc{UnFma}\ \star \in \{\neg, \bullet_I, \circ_I, \blacklozenge_I, \lozenge_I, \blacksquare_I, \square_I, \blacktriangleleft_I, \triangleright_I\} \qquad \frac{\Delta;\ \Gamma \vdash \varphi_1 \qquad \Delta;\ \Gamma \vdash \varphi_2}{\Delta;\ \Gamma \vdash \varphi_1 \star \varphi_2}\ \textsc{BinFma}\ \star \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \mathsf{S}_I, \mathsf{U}_I\}$$

$$\frac{}{\Delta;\ \Gamma \vdash \cdot}\ \textsc{Wild} \qquad \frac{\Delta;\ \Gamma \vdash r}{\Delta;\ \Gamma \vdash \star r}\ \textsc{UnRex} \quad \star \in \{?, *\} \qquad \frac{\Delta;\ \Gamma \vdash r_1 \qquad \Delta;\ \Gamma \vdash r_2}{\Delta;\ \Gamma \vdash r_1 \star r_2}\ \textsc{BinRex} \quad \star \in \{+,\ \}$$

$$\frac{\Delta;\ \Gamma, v_1 : \tau_1, \ldots, v_n : \tau_n \vdash \varphi}{\Delta;\ \Gamma \vdash \exists v_1, \ldots, v_n.\ \varphi}\ \textsc{Exists} \qquad \frac{\Delta;\ \Gamma, v_1 : \tau_1, \ldots, v_n : \tau_n \vdash \varphi}{\Delta;\ \Gamma \vdash \forall v_1, \ldots, v_n.\ \varphi}\ \textsc{Forall}$$

$$\frac{\Delta;\ v_1 : \tau_1, \ldots, v_n : \tau_n \vdash \varphi_1 \qquad \Delta, (p, (\tau_1, \ldots, \tau_n));\ \Gamma \vdash \varphi_2}{\Delta;\ \Gamma \vdash \mathsf{let}\ p(v_1, \ldots, v_n) = \varphi_1\ \mathsf{in}\ \varphi_2}\ \textsc{Let ... in}$$

$$\frac{\Gamma \vdash v_1 :: \tau \qquad \Gamma, \bar{z}s : \bar{\sigma} \vdash v_2 :: \tau \qquad \Delta;\ \Gamma, \bar{z}s : \bar{\sigma} \vdash \varphi \qquad \tau \in \mathsf{Num} \qquad \bar{z}s = \mathsf{fv}(\varphi) - \bar{v}s}{\Delta;\ \Gamma \vdash v_1 \leftarrow \mathsf{SUM}\ v_2;\ \bar{v}s\ \varphi}\ \textsc{Sum}$$

$$\frac{\Gamma \vdash v_1 :: \mathsf{Z} \qquad \Gamma, \bar{z}s : \bar{\sigma} \vdash v_2 : \tau \qquad \Delta;\ \Gamma, \bar{z}s : \bar{\sigma} \vdash \varphi \qquad \bar{z}s = \mathsf{fv}(\varphi) - \bar{v}s}{\Delta;\ \Gamma \vdash v_1 \leftarrow \mathsf{CNT}\ v_2;\ \bar{v}s\ \varphi}\ \textsc{Cnt}$$

$$\frac{\Gamma \vdash v_1 :: \mathsf{F} \qquad \Gamma, \bar{z}s : \bar{\sigma} \vdash v_2 : \tau \qquad \Delta;\ \Gamma, \bar{z}s : \bar{\sigma} \vdash \varphi \qquad \tau \in \mathsf{Num} \qquad \bar{z}s = \mathsf{fv}(\varphi) - \bar{v}s}{\Delta;\ \Gamma \vdash v_1 \leftarrow A\ v_2;\ \bar{v}s\ \varphi}\ \textsc{AvgMed} \quad A \in \{\mathsf{AVG}, \mathsf{MED}\}$$

$$\frac{\Gamma \vdash v_1 :: \tau \qquad \Gamma, \bar{z}s : \bar{\sigma} \vdash v_2 : \tau \qquad \Delta;\ \Gamma, \bar{z}s : \bar{\sigma} \vdash \varphi \qquad \tau \in \mathsf{Ord} \qquad \bar{z}s = \mathsf{fv}(\varphi) - \bar{v}s}{\Delta;\ \Gamma \vdash v_1 \leftarrow A\ v_2;\ \bar{v}s\ \varphi}\ \textsc{MinMax} \quad A \in \{\mathsf{MIN}, \mathsf{MAX}\}$$

Figure 4.2: Well-formed rules for MFODL formulas

4.2 Type inference

One important feature of CMFODL's type system is its type inference capability. It allows users to write formulas without needing to annotate types of variables and predicate arguments. For example, let us look at the formula $\mathsf{x.f} > \mathsf{a} \wedge \mathsf{S}(x)$ and the type inference rules of Figure 4.1 and 4.2. The type checker may first learn from the sub-formula $\mathsf{x.f} > \mathsf{a}$ that x is of an instance of some product type τ_1, declaring a field named f of type $\tau_2 \in \mathsf{Ord}$. This can be derived by applying the rules Proj from Figure 4.1 and OrdRel from Figure 4.2. It further learns that the variable a is of type τ_2 too, based on the rule OrdRel. The type checker eventually encounters the sub-formula $\mathsf{S}(\mathsf{x})$. Using inference rule Sort, it learns that x is in fact of type $\mathsf{S}\ \{\ FS\ \}$ and therefore $\tau_1 = \mathsf{S}\ \{\ FS\ \}$. Finally, the formula is well-typed as long as FS declares a field f of type τ_2.

The example above should provide an intuition about the problem that CMFODL's type inference tries to solve. We now formalize this process. First, we need to introduce some new definitions:

Definition 4.1 Constant types

We formalize a constant type τ as the set of constant values associated with τ. Therefore, the type Str represents the infinite set of all string constants, whereas the type $\bot$ represents the empty set. We call a a subtype of b whenever $a \subseteq b$ holds. The set of all constant types is defined as $\mathsf{TCst} := \{\mathsf{Z}, \mathsf{F}, \mathsf{Str}, \mathsf{RE}, \mathsf{B}, \bot\} \cup \mathsf{N}\ \{\ FS\ \}$, where $\mathsf{N}\ \{\ FS\ \}$ ranges over all custom product types. Additionally, we declare all constant types in TCst as pairwise disjunct: $\forall (o,s) \in \mathbb{D}, t, t' \in \mathsf{TCst}.\ s \in t \wedge s \in t' \implies t = t'$.

Definition 4.2 Type classes

We redefine a type class as a non-empty set of types with the additional constraint, that whenever $a \in \mathsf{A}$ holds for some type a and some type class A, $\forall\ b \subseteq a.\ b \in \mathsf{A}$ holds. Next, we declare the set of all type classes as $\mathsf{Cls} := \{\mathsf{Any}, \mathsf{Num}, \mathsf{Ord}, \mathsf{Prod}[\mathcal{C}]\}$. The newly introduced type class $\mathsf{Prod}[\mathcal{C}]$ is parametric on a set $\mathcal{C}$ of constraints on the fields of its product types. We define the elements of this type class in more detail below.

Definition 4.3 Symbolic types

We define a symbolic type $\mathsf{TSymb}[\mathsf{C}] \subseteq \{(\mathsf{C}, i) \mid i \in \mathbb{N}, \mathsf{C} \in \mathsf{Cls}\}$ as a set of tuples consisting of a type class and a unique index: $\forall (\mathsf{A}, i), (\mathsf{B}, j) \in \mathsf{TSymb} : i = j \implies \mathsf{A} = \mathsf{B}$. The set $\mathsf{TSymb} := \bigcup_{\mathsf{C} \in \mathsf{Cls}} \mathsf{TSymb}[\mathsf{C}]$ represents the set of all symbolic types. Based on these definitions, we declare the set $\mathsf{Ty} := \mathsf{TCst} \cup \mathsf{TSymb}$, which consists of all constant and symbolic types.

Definition 4.4 The more precise type relation $\preceq$

We define the relation $a \preceq b$. It is read as *"type a is more or equally precise than type b"*. It is defined on constant types a,b as $a \subseteq b$ and can be lifted to symbolic types as $(\mathsf{A},i) \preceq (\mathsf{B},j) \iff \mathsf{A} \subset \mathsf{B} \vee \mathsf{A} = \mathsf{B} \wedge i \leq j$, and to a relation between a constant type a and a symbolic type (C,i) as $a \preceq (\mathsf{C},i) \iff a \in \mathsf{C}$.

Proposition 4.5 *$\preceq$ is a partial order relation.*

Proof In the case of both operands being constant types, $\preceq$ is equivalent to $\subseteq$ and therefore, the statement holds. Furthermore, $\forall (A,i) \in \mathsf{TSymb}.\ (A,i) \preceq (A,i) \equiv A \subset A \vee A = A \wedge i \leq i$ holds, and therefore, $\preceq$ is reflexive. In a similar manner, we prove that $\preceq$ is anti-symmetric by showing that $(A,i) \preceq (B,j) \wedge (B,j) \preceq (A,i) \implies (A,i) = (B,j)$ holds for all $A,B \in \mathsf{TSymb}$:

$$
\begin{aligned}
& (A,i) \preceq (B,j) \wedge (B,j) \preceq (A,i) \\
\iff & (A \subset B \vee A = B \wedge i \leq j) \wedge (B \subset A \vee B = A \wedge j \leq j) && \text{(def. } \preceq\text{)} \\
\iff & (A \subset B \wedge B \subset A) \vee (A \subset B \wedge B = A \wedge j \leq i) \vee \\
& (A = B \wedge i \leq j \wedge B \subset A) \vee \\
& (A = B \wedge i \leq j \wedge B = A \wedge i \leq j) && \text{(distrib. of } \vee\text{)} \\
\implies & (A = B \wedge i \leq j \wedge B = A \wedge i \leq j) \\
\iff & (A = B \wedge i = j) \iff (A,i) = (B,j)
\end{aligned}
$$

For both reflexivity and anti-symmetry, we do not need to cover mixed operands: By definition, a symbolic type is never more precise than a constant type. To show that $\preceq$ is transitive, we therefore need to cover following cases for $a,b \in \mathsf{TCst}$, $(A,i),(B,j),(C,k) \in \mathsf{TSymb}$:

$$
\begin{aligned}
& \textbf{case } a \preceq b \preceq (A,i) \\
& \iff a \subseteq b \wedge b \in A \\
& \implies a \in A && \text{(def. type class)} \\
& \iff a \preceq (A,i) && \text{(def. } \preceq\text{)} \\
& \textbf{case } a \preceq (A,i) \preceq (B,j) \\
& \iff a \in A \wedge (A \subset B \vee A = B \wedge i \leq j) \\
& \implies a \in B \\
& \iff a \preceq B && \text{(def. } \preceq\text{)} \\
& \textbf{case } (A,i) \preceq (B,j) \preceq (C,k) \\
& \iff (A \subset B \vee A = B \wedge i \leq j) \wedge (B \subset C \vee B = C \wedge j \leq k) \\
& \iff (A \subset B \wedge B \subset C) \vee (A \subset B \wedge B = C \wedge j \leq k) \vee \\
& \quad (A = B \wedge i \leq j \wedge B \subset C) \vee \\
& \quad (A = B \wedge i \leq j \wedge B = C \wedge j \leq k) && \text{(distrib. of } \vee\text{)} \\
& \implies A \subset C \vee A = C \wedge i \leq k \\
& \iff (A,i) \leq (C,k) && \text{(def. } \preceq\text{)}\ \square
\end{aligned}
$$

The proof above implies that the set of all types Ty, together with the relation $\preceq$, forms a partially ordered set. Using the relation $\preceq$, we can now declare the type class $\mathsf{Prod}[\mathcal{C}]$ more precisely: $\mathsf{Prod}[\mathcal{C}] := \{\mathsf{N}\{\ FS\ \} \mid \forall (f_c : t_c) \in \mathcal{C} : \exists \{f : t\} \in FS.\ f = f_c \wedge t \preceq t_c\}$ for a set of constraints $\mathcal{C} \subseteq \mathsf{N} \times \mathsf{Ty}$. Intuitively, $\mathcal{C}$ is the minimal set of fields and their symbolic or constant types, which a product type must consist of to be an element of $\mathsf{Prod}[\mathcal{C}]$.

Proposition 4.6 Ty *together with* $\preceq$ *forms a meet-semilattice.*

Proof We first prove the existence of a unique greatest and smallest element in the partially ordered set Ty: We recall that Ty contains the constant type $\bot$ representing the empty set, and the symbolic type (Any, i_{max}), with Any representing the type class containing all types, and i_{max} equal the largest index of all symbolic types of class Any: $\forall (A, j) \in \mathsf{TSymb} : A = \mathsf{Any} \implies j \leq i$. To prove the existence of a unique greatest and smallest element in Ty, it is sufficient to show that $t \preceq (\mathsf{Any}, i_{max})$ and $\bot \preceq t$ hold for all $t \in \mathsf{Ty}$:

In case of $t \in \mathsf{TCst}$, then $t \preceq \mathsf{TSymb}[\mathsf{Any}] \equiv t \in \mathsf{Any}$ holds by definition of Any. $\bot \preceq t \equiv \bot \subseteq t$ is also true by definition of $\bot$.

In case of $t = (A, i) \in \mathsf{TSymb}$, $(A, i) \preceq (\mathsf{Any}, i_{max})$ is equivalent to $A \subset \mathsf{Any} \vee A = \mathsf{Any} \wedge i \leq i_{max}$, proving the existence of a greatest element. Finally, $\bot \preceq t = (A, i)$ is equivalent to $\bot \in A$ and holds based on the fact that the type class A is non-empty and contains the transitive closure over all its members, including the empty set. The uniqueness of both the greatest and smallest element can be proven by contradiction: Assume there exist two greatest elements t_1, t_2 of Ty: By definition we have $t_1 \preceq t_2$ and $t_2 \preceq t_1$, which – by definition of $\preceq$ – can only hold if and only if $t_1 = t_2$. We can similarly prove that the smallest element is unique.

We now show that for every subset $\{t_1, t_2\} \subseteq \mathsf{Ty}$ of size two, there exists a unique greatest lower bound, representing the meet of t_1 and t_2: Because all types in TCst are defined as pairwise disjunct, the definition of $\preceq$ infers that the only type more precise than any constant type is the bottom type: $\forall t, t' \in \mathsf{TCst}.\ t \preceq t' \implies t = t' \vee t = \bot$. Therefore, if $t_1, t_2 \in \mathsf{TCst}$, their meet is uniquely represented by $\bot$.
In case of $t_i \in \mathsf{TCst}$, $t_j \in \mathsf{TSymb}$ for $i \in \{0, 1\}$, $j = 2 - i$: If $t_i \preceq t_j$, their meet is clearly t_i. In any other case, the only existing lower bound of t_i and t_j is $\bot$, and therefore equal to their meet.
For the last case where $t_1, t_2 \in \mathsf{TSymb}$, we distinguish between the following cases: If $t_1, t_2 \in \mathsf{TSymb} - \mathsf{Prod}[\mathcal{C}]$ for some $\mathcal{C}$, we can derive the existence of a meet by the fact that $(\mathcal{P}(\mathsf{TCst}), \subseteq)$ forms a complete lattice. If exactly one of t_1, t_2 is part of $\mathsf{Prod}[\mathcal{C}]$ for some $\mathcal{C}$, then their common meet is $\bot$, because Prod is the only type class containing any product types. Finally, if $t_1 \in \mathsf{Prod}[\mathcal{C}_1]$, $t_2 \in \mathsf{Prod}[\mathcal{C}_2]$, we derive their meet $\mathsf{Prod}[\mathcal{C}_{meet}]$ by merging the constraints $\mathcal{C}_1$ and $\mathcal{C}_2$ the following way: $\mathsf{Prod}[\mathcal{C}_{1 \setminus 2}] = \{(f, t) \in \mathcal{C}_1 \mid \nexists (f', t') \in \mathcal{C}_2.\ f' = f\}$, $\mathsf{Prod}[\mathcal{C}_{2 \setminus 1}] =$

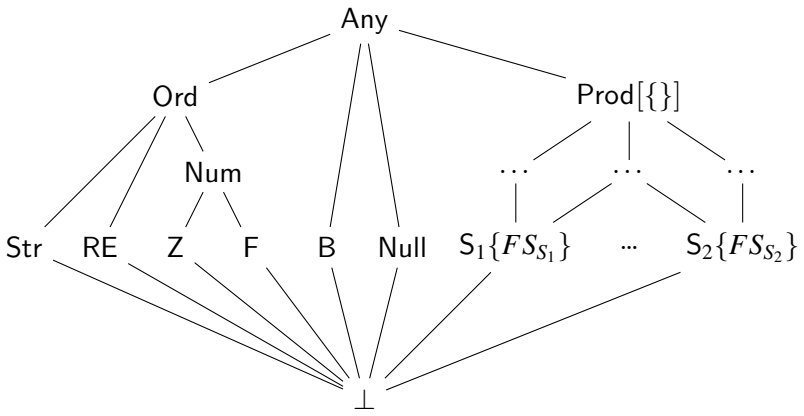

Figure 4.3: Simplifed Hasse diagram of MonPoly 's type system

$\{(f,t) \in \mathcal{C}_2 \mid \nexists (f',t') \in \mathcal{C}_1.\ f' = f\}$, $\mathcal{C}_{common} = \{f \in \mathsf{N} \mid (f,\_) \in \mathcal{C}_1 \wedge (f,\_) \in \mathcal{C}_2\}$ and $\mathcal{C}_{1\sqcap 2} = \{(f, \mathcal{C}_1[f] \sqcap \mathcal{C}_2[f]) \mid f \in \mathcal{C}_{common}\}$, where $\sqcap$ is the meet operator. We can then define $\mathcal{C}_{meet} = \mathcal{C}_{1\setminus 2} \cup \mathcal{C}_{2\setminus 1} \cup \mathcal{C}_{1\sqcap 2}$. □

Using our definitions above, we can visualize the type lattice used by MonPoly by a simplified Hasse diagram in Figure 4.2. Symbolic types are represented by their type class. The symbolic type represented by $\mathsf{Prod}[\{\}]$ stands for all product types because the set of constraints $\mathcal{C}$ is empty.

Algorithm 1 shows the conceptual structure of MonPoly 's type inference algorithm. The key part of the algorithm is in line 18, where it computes the meet of the actual and expected type of a term. Because Ty together with $\preceq$ forms a lattice, the meet is guaranteed to exist. Whenever the meet is equal to $\bot$, the type inference algorithm throws an error. We do not regard variables of type $\bot$ as semantically valid: There exists no value assignable to $\bot$, as it represents the empty type set. Finally, after type checking an input formula f, all variables are expected to have a constant type assigned: if a variable is of a symbolic type after type checking, the formula is polymorphic as the inference algorithm was unable to derive its type.

Intuitively, every new variable gets assigned a type at the top of the Hasse diagram. With every new constraint propagated through the symbol table, the type of a variable may *move* downwards along the lines of the Hasse diagram.

Algorithm 1: MonPoly 's type inference algorithm

Data: input formula f, predicate schema Δ, custom sorts $\mathbf{T}$, symbol table Γ

`/* initialize all variables of the current symbol table with Any: */`

foreach $v \in \Gamma$ **do**
  $i \leftarrow$ unique index;
  $v \leftarrow (\mathsf{Any}, i)$;
end

`/* recurse over input formula: */`

foreach $f' \in walk(f)$ **do**
  foreach $t \in terms(f')$ **do**
    `/* Apply the rules from Figure 4.2: */`
    $\tau_{expt} \leftarrow$ expected type for term t to make f' well-formed;
    $typecheck\_term(t, \tau_{exp})$;
  end
end

`/* Verify that the type of all variables has been resolved to a constant type: */`

foreach $v \in \Gamma$ **do**
  if *type of* $v \in \mathsf{TSymb}$ **then**
    throw *type error: unresolved type*
  end
end

Function $typecheck\_term(t, \tau_{exp})$ **is**
  `/* Apply the rules from Figure 4.1: */`
  $t_{curr} \leftarrow$ type of t based on current Γ;
  $\tau_{new} \leftarrow$ meet of $\{\tau_{curr}, \tau_{exp}\}$;
  if $\tau_{new} = \bot$ **then**
    throw *type error: incompatible types*
  else
    update every occurence of $\{t_{curr}, t_{exp}\}$ with t_{new} in Γ;
  end
end

Chapter 5

Compiling CMFODL

The complex-typed syntax extensions described in Chapter 3 are not yet supported by MonPoly 's monitoring algorithm. It has no support for custom sorts, and instead only supports predicate symbols over the primitive sorts $\mathsf{PS} = \{\mathsf{Z}, \mathsf{F}, \mathsf{Str}, \mathsf{RE}\}$. For MonPoly to support the full CMFODL syntax described in Chapter 3, the extension presented in this work performs the following two steps, which preserve the semantics of the original formula: We first transform the temporal structure ρ to a new temporal structure ρ^P over finite relations, which domains range over primitive sorts. This transformation is described in Section 5.1 and 5.2. We then replace unsupported syntactic elements by compiling CMFODL formulas to MFODL formulas. This is described in Section 5.3. These steps are implemented in MonPoly itself and are carried out before and during the actual monitoring, allowing users to write MFODL formulas against complex typed temporal structures without any manual transformations.

5.1 Transforming signatures

A signature $\mathcal{S}^P = (C^P, R^P, \iota)$, known to MonPoly 's monitoring algorithm, consists of a finite set of constant symbols C^P, a finite set of predicates R^P disjoint from C^P, and a function $\iota : R^P \to \mathbb{N}$ assigning each predicate $r \in R^P$ an arity $\iota(r)$ [5].

We map a signature $\Delta = (S, S_{def}, C)$ described in Section 3.2 to a predicate signature $\mathcal{S}^P = (C^P, R^P, \iota)$ with $R^P = \mathsf{CS}$, $\iota := r \mapsto |S_{def}(r)| + 1$ and $C^P = C$, where $|S_{def}(r)|$ corresponds to the number of fields of the custom sort r and making the assumption that C contains only values of primitive sorts. In other words, we create a predicate for each custom sort with an arity equal to the number of fields, plus an additional argument used as an object identifier introduced below in Section 5.2. We further create a relation schema $\mathsf{RS} := r \mapsto \mathsf{PS}^{\iota(r)}$ from Δ, mapping each predicate in R^P to the sorts of its arguments. This

process is described by Algorithm 2. The sorts of the arguments correspond to the sort of each field of the custom sort r. Fields whose values are of custom sorts themselves represent nested objects and are encoded as a reference pointer to a tuple of another relation.

Example 5.1 We define the signature $\Delta = (\mathsf{PS} \cup \{\mathsf{User}, \mathsf{Request}, \mathsf{Report}\}, \{\mathsf{User} \mapsto \{\mathsf{name} : \mathsf{Str}\}, \mathsf{Request} \mapsto \{\mathsf{url} : \mathsf{Str}, \mathsf{user} : \mathsf{User}\}, \mathsf{Report} \mapsto \{\mathsf{reason} : \mathsf{Str}, \mathsf{user} : \mathsf{User}\}\}, \emptyset)$. The corresponding predicate schema is $\mathcal{S}^P = (\emptyset, \{\mathsf{User}, \mathsf{Request}, \mathsf{Report}\}, \{\mathsf{User} \mapsto 2, \mathsf{Request} \mapsto 3, \mathsf{Report} \mapsto 3\})$. From Δ, Algorithm 2 will generate the relation schema $\mathsf{RS} = \{\mathsf{User} \mapsto (\mathsf{Z}, \mathsf{Str}), \mathsf{Request} \mapsto (\mathsf{Z}, \mathsf{Str}, \mathsf{Z}), \mathsf{Report} \mapsto (\mathsf{Z}, \mathsf{Str}, \mathsf{Z})\}$.

Algorithm 2: Generate relation schema from Δ

Data: signature $\Delta = (S, S_{def}, C)$
Result: relation schema $\mathsf{RS} = r \mapsto \mathsf{PS}^{\iota(r)}$

```
1  foreach r ∈ S \ PS do
       /* loop over all fields of custom sort r:                    */
2      foreach (i, f) ∈ enumerate(ran(S_def(r))) do
3          if S_def(r, f) ∈ PS then
4              t_i ← S_def(r, f);
5          else
6              t_i ← Z;
7          end
8      end
       /* first entry corresponds to identifier:                    */
9      RS [r ↦ (Z, t_1, ..., t_|ran(S_def(r))|)];
10 end
```

5.2 Transforming complex-typed temporal structures

A temporal structure ρ of the form $(\tau_i, \mathcal{D}_i)_{i \in \mathbb{N}}$ with $\mathcal{D}_i = (\mathbb{D}, \mathbb{C}, \mathbb{O}_i)$, as described in Section 3.2, is not processable by MonPoly 's monitoring algorithm. It must be transformed to a temporal structure ρ^P of the form $(\tau_i, \mathcal{D}_i^P)_{i \in \mathbb{N}}$, where $\mathcal{D}_i^P$ is a structure over a signature $\mathcal{S}^P = (C^P, R^P, \iota)$ as defined in Section 5.1. It consists of a domain $|\mathcal{D}^P| = \{d \mid (s, d) \in \mathbb{D} \wedge s \in \mathsf{PS}\}$ and interpretations $c^{\mathcal{D}^P} \in |\mathcal{D}^P|$ and $r^{\mathcal{D}_i^P} \subseteq |\mathcal{D}^P|^{\iota(r)}$ for each $c \in C^P$ and $r \in R^P$ [5]. The set $r^{\mathcal{D}_i^P}$ is constructed as described in Algorithm 3: Values of custom sorts are stored in their dedicated relations, while unique identifiers act as a reference from a tuple entry of a custom sort to its value stored as a tuple in another relation.

One important distinction must be made regarding the uniqueness of the identifiers assigned to objects in Algorithm 3, line 5: In this implementation,

identifiers are unique per time point, meaning that unrelated objects of two different time points may share the same identifier. This has a major impact on how formulas must be preprocessed before compilation, to retain their original semantics. Section 9.1 provides a rationale for this decision.

Example 5.2 Let us again consider signatures Δ, $\mathcal{S}^P$ and RS from example 5.1 on page 21. We additionally define the following temporal structure ρ, where @10 refers to the timestamp of the time point, and the right-hand side refers to $\mathbb{O}_i$:

$$
\begin{aligned}
@10: \quad & (\mathsf{Request}, \{\mathsf{url} : (\mathsf{Str}, \text{secr.et}), \mathsf{user} : (\mathsf{User}, \{\mathsf{name} : (\mathsf{Str}, \text{Eve})\})\}), \\
& (\mathsf{Request}, \{\mathsf{url} : (\mathsf{Str}, \text{ethz.ch}), \mathsf{user} : (\mathsf{User}, \{\mathsf{name} : (\mathsf{Str}, \text{Alice})\})\}) \\
@20: \quad & (\mathsf{Report}, \{\mathsf{reason} : (\mathsf{Str}, \text{NoAuth}), \mathsf{user} : (\mathsf{User}, \{\mathsf{name} : (\mathsf{Str}, \text{Alice})\})\})
\end{aligned}
$$

Using Algorithm 3 we can generate the temporal structure ρ^P:

$$
\begin{aligned}
@10: \quad & \{\mathsf{User}(1, \text{Eve}), \mathsf{Request}(2, \text{secr.et}, 1), \mathsf{User}(3, \text{Alice}), \mathsf{Request}(4, \text{ethz.ch}, 3)\} \\
@20: \quad & \{\mathsf{User}(1, \text{Alice}), \mathsf{Report}(2, \text{NoAuth}, 1)\}
\end{aligned}
$$

Note that in ρ^P, two tuples exist in the relation User with the same identifier 1, referring to two unrelated users Alice and Eve. Because they appear at different time points, they can share the same identifier.

Algorithm 3: Generate tuples for time point i

Data: objects $\mathbb{O}_i$ of time point i
Result: $r^{\mathcal{D}_i^P}$ for all $r \in \{s \mid (s,\_) \in \mathbb{O}\downarrow\}$

foreach $o = (s,f) \in \mathbb{O}$ *with* $s \in \mathsf{CS}$ **do**
  call *add_tuple(o)*;
end
Function *add_tuple(o)* **is**
  Data: an object $o \in \mathbb{O}\downarrow$
  Result: a unique identifier assigned to the tuple of o
  $id \leftarrow$ unique identifier;
  if $o = (s, \Pi f)$, *for some* $s \in \mathsf{CS}$ **then**
    foreach $(i,n) \in \mathrm{enumerate}(\mathrm{dom}(f))$ **do**
      $(s_n, v_n) \leftarrow f(n)$;
      $t_i = \begin{cases} v_n & \text{if } s_n \in \mathsf{PS} \\ add\_tuple(v_n) & \text{else} \end{cases}$;
    end
    `/* Add new tuple to relation` s`: */`
    $s^{\mathcal{D}_i^P} = s^{\mathcal{D}_i^P} \cup \left\{ \left(id, t_1, \ldots, t_{|\mathrm{dom}(f)|} \right) \right\}$;
  end
  else if $o = (s,v)$ *for some* $s \in \mathsf{PS}$ **then**
    $s^{\mathcal{D}_i^P} = s^{\mathcal{D}_i^P} \cup \{(id, v)\}$
  end
  return *id*;
end

5.3 Compiling MFODL formulas

This section discusses the compilation of a CMFODL formula evaluated over a temporal structure ρ, to an MFODL formula evaluated under ρ^P. We distinguish two separate steps: A preprocessing step described in Section 5.3.1 and a subsequent compilation step described in Section 5.3.2. The purpose of the preprocessing is to prepare the formula for the compilation step, such that the compiled formula retains the semantics of the input formula in the context of the predicate signature $\mathcal{S}^P$ and structure $\mathcal{D}^P$ described in previous sections 5.1 and 5.2. Technically, a complex-typed MFODL formula may be compiled without prior preprocessing, but it may change its semantics. The preprocessing therefore depends strongly on the transformation of the temporal structure described in Section 5.2, primarily how identifiers are assigned to objects.

We declare the function $\mathsf{typeof}_\Gamma : \mathsf{Terms} \to \mathsf{Ty}$, mapping a given term to its type under the current symbol table Γ, as inferred by the type inference algorithm

described in Chapter 4. Next, we define the set of field terms $\mathbb{F}_t$ for some term t. Because MFODL only allows assigning terms to variables, we can assume that t is a variable or projection term without loss of generality. Furthermore, custom sorts are constrained to be non-recursive, as introduced in Section 3.1. Therefore, the following sets $\mathbb{F}_t$ and $\mathbb{L}_t$ are guaranteed to be well defined and of finite size for any given term t.

$$\mathbb{F}_v = \begin{cases} \{v\} & \text{if } \mathsf{type\_of}_\Gamma(v) \in \mathsf{PS} \\ \{v.n \mid (n,t) \in S_{def}(t)\} \cup \left\{v.f \mid f \in \bigcup_{(n,t)\in S_{def}(t)} \mathbb{F}_{v.n}\right\} & \text{else} \end{cases}$$

Similarly, we define the set of leaf terms $\mathbb{L}_t$ for some term t:

$$\mathbb{L}_v = \begin{cases} \{v\} & \text{if } \mathsf{type\_of}_\Gamma(v) \in \mathsf{PS} \\ \left\{v.f \mid f \in \bigcup_{(n,t)\in S_{def}(t)} \mathbb{L}_{v.n}\right\} & \text{else} \end{cases}$$

Example 5.3 Given custom sorts $\mathsf{Request}\{\mathsf{url}:\mathsf{Str}, \mathsf{user}:\mathsf{User}\}$ and $\mathsf{User}\{\mathsf{name}:\mathsf{Str}\}$, and a variable t with $\mathsf{type\_of}_\Gamma(t) = \mathsf{Request}$, the set $\mathbb{F}_t$ is equal to $\{r.url, r.user, r.user.name\}$, while $\mathbb{L}_t$ is equal to $\{r.url, r.user.name\}$.

We overload the definition of $\mathbb{L}_c$ for some constant value c, such that $\mathbb{L}_c$ consists of the leaf values of c instead:
Given a constant $c = \mathsf{Request}\{\mathsf{url}:\text{"}url.tld\text{"}, \mathsf{user}:\{\mathsf{name}:\text{"}alice\text{"}\}\}$, $\mathbb{L}_c$ is equal to {"*url.tld*","*alice*"}.

5.3.1 Semantic-preserving preprocessing

This section introduces transformations that preserve the semantics of an input formula after compilation. Because Algorithm 3 assigns locally unique identifiers to objects, certain formula structures must be rewritten before compilation. In general, we must not pass an identifier referencing an object over a temporal operator: The same identifier may reference another unrelated object in a future or past time point. Similarly, we must expand structural equality between two values of complex sorts: Comparing the values of two identifiers themselves may lead to unexpected behavior whenever the identifiers have been assigned at two different time points.

To illustrate this problem, we refer to Example 5.2 on page 22 and declare the formula $\varphi = \mathsf{Request}(\mathsf{u}) \wedge \mathsf{r.url} = \mathrm{secr.et} \rightarrow \Diamond_{[0,50]}\, \exists \mathsf{u}.\ \mathsf{Report}(\mathsf{u}) \wedge \mathsf{u.user} = \mathsf{r.user}$. When interpreting the formula φ over the temporal structure ρ^P of Example 5.2 without preprocessing, the formula will evaluate as true. However, by examining the example more carefully, one may realize that the Report at timestamp @20 is addressed to *Alice* instead of *Eve*, who sent the malicious request. The issue arises because the user *Alice* of the Report at timestamp @20 shares the same identifier with the unrelated user *Eve* from timestamp @10.

The following paragraphs describe each transformation applied during the preprocessing.

Preprocessing of equalities We interpret MFODL's equality operator on operands of custom sorts as structural equivalence. Section 9.1 discusses this decision in more detail. Hence, a formula of the form $\varphi \equiv a = b$, where both a and b are terms of the same custom sort s, can be transformed to a semantically equivalent formula ψ, where all fields on both values are compared separately: $\psi \equiv \bigwedge_{(f_1,f_2)\in(\mathbb{L}_a,\mathbb{L}_b)} f_1 = f_2$. Notice that the rule EQUAL, introduced in Figure 4.2, does enforce the same sort on both sides of an assignment, which implies that $\mathbb{L}_a$ and $\mathbb{L}_b$ contain the same projections, but with different prefixes.

Referring to Example 5.3 on page 24, comparing two values a, b of custom sort Request translates to the formula $\psi \equiv a.url = b.url \wedge a.user.name = b.user.name$.

Preprocessing of let-statements The domain of predicates introduced by a let-statement may range over complex sorts. We introduce a transformation that flattens all predicate arguments a of a complex sort by replacing a with a subset of its leaves $\mathbb{L}_a$. To formalize this process, we first define the set of usages $\mathbb{U}_f[v] \subseteq \mathsf{Terms}$ as a set of variables and projections that make use of a given variable v in the formula f. To transform a formula of the form $\mathsf{let}\ n(\vec{v}) = \psi\ \mathsf{in}\ \varphi$ for some formulas φ and ψ, we first define the usage $\mathbb{U}_\psi[a_i]$ for every argument $a_i \in \vec{v}$. Next, we define a mapping $nv : \mathsf{Terms} \to \mathsf{V}$, mapping each usage $u \in \mathbb{U}_\psi[a_i]$ to a unique new variable $v_u \notin \mathsf{fv}(\psi)$. We make use of nv to construct the formula ψ' by replacing each usage u in ψ with $nv(u)$. Finally, we construct φ' by transforming each usage of predicate n in φ: Each argument term t_i in $n(t_1,\ldots,t_k)$ is replaced by $\vec{t_i} = \{l.f \in \mathbb{L}_{t_i} \mid a_i.f \in \mathbb{U}_\psi[a_i]\}$. The resulting formula of the transformation is $\mathsf{let}\ n\left(\bigcup_{i\in[1,k]} \vec{t_i}\right) = \psi'\ \mathsf{in}\ \varphi'$.

Example 5.4 Reaching back to the signatures defined in Example 5.3 on page 24, transforming the formula $\varphi \equiv \mathsf{let}\ \mathsf{p}(\mathsf{r}) = \mathsf{r.url} = \text{"url.tld"} \wedge \mathsf{r.user.name} = \text{"alice"}\ \mathsf{in}\ \mathsf{Request}(\mathsf{r}) \wedge \mathsf{p}(\mathsf{r})$ results in a semantically equivalent formula $\psi \equiv \mathsf{let}\ \mathsf{p}(\mathsf{a},\mathsf{b}) = \mathsf{a} = \text{"url.tld"} \wedge \mathsf{b} = \text{"alice"}\ \mathsf{in}\ \mathsf{Request}(\mathsf{r}) \wedge \mathsf{p}(\mathsf{r.url},\mathsf{r.user.name})$.

5.3.2 Compilation to MFODL

This section describes the compilation process to form an MFODL formula from a complex typed formula, such that the compiled formula retains its semantics according to the temporal structure $\mathcal{D}^P$. We look at each affected type of formula separately:

Compilation of custom sorts and projections We first realize that certain MFODL operators defined in Figure 3.2, such as quantifiers, aggregations, and let-statements, create a new variable scope by introducing mappings to the

valuation v for new variables. If a newly introduced mapping overwrites an existing mapping, we denote the original mapping as shadowed. We call the formula f' the scope of a variable a whenever the mapping for a in v is initially introduced in f'. The scope of a free variable of an input formula f ranges over the whole formula f. We also declare Λ_f^φ to be the set of all subformulas of φ of the form f.

Referring to the operational semantics of MFODL declared in Figure 3.2, the formula $s(t)$ for some custom sort s is satisfied at time point i of temporal structure $\rho = (\tau_i, \mathcal{D}_i)_{i\in\mathbb{N}}$ whenever $v(t) \in \mathbb{O}^{\mathcal{D}_i}$ is satisfied. For every scope φ of variable t, consisting of one or multiple subformulas of the form $s(t)$ and s equal to the sort of t, we propose a semantically equivalent formula $\psi \equiv \exists \vec{v}.\ \chi$ such that $\delta, v, i \models_\rho \varphi \iff \delta, v, i \models_{\rho^P} \psi$ is satisfied, where ρ^P refers to the transformed temporal structure from Section 5.2, $\vec{v}$ refers to a vector of variables, and χ to a new formula defined below. We only consider cases where t refers to a variable term. Any other type of term can be assigned to a variable first.

To define $\vec{v}$, we declare a function $path : \mathsf{Terms} \to \mathsf{V}$ that maps a projection term to a variable by the following rules, where $\cdot$ is used as concatenation operator such that the resulting variable for a given tuple (o, f) is unique in V:

$$\begin{aligned} path(o.f) &= path(o) \cdot f && \text{for some projection term } o.f \\ path(v) &= v && \text{for some variable term } v \end{aligned}$$

$\vec{v}$ can then be described as $\left[\bigcup_{s(t)\in\Lambda^\varphi_{s(t)}} \mathbb{F}_t\right]$ ' $\lambda p.\ path(p)$, where we first gather the projections of the recursive fields of each variable t of all subformulas of the form $s(t)$ in φ, and finally map them to a set of variables using the mapping $path$ defined above.

For the definition of χ, we first define a mapping g from $(\mathsf{V} \times \mathsf{CS})$ to a predicate formula: $(v, s) \mapsto s\,(v, path(v.f_1), \ldots, path(v.f_k))$ for field names $f_1, \ldots f_k \in \mathsf{dom}(S_{def}(s))$, where $s \in R^P$ and the number of arguments of s corresponds to $\iota(s)$. In other words, given a custom sort $\mathsf{S}\{f_1 : \mathsf{Str}, f_2 : \mathsf{Z}\}$ and a variable a, the resulting formula of $g(S, a)$ is $S(a, a \cdot f_1, a \cdot f_2)$. We also define the set of predicate formulas $\mathbb{P}_s[v]$ of a custom sort s and variable v, as $\mathbb{P}_s[v] = \{g(v, s)\} \cup \bigcup_{(n,t)\in S_{def}(s) \text{ where } t\in\mathsf{CS}} \mathbb{P}_t[v \cdot n]$. To construct χ, we replace every occurrence of subformulas of the form $s(t)$ for a custom sort s in φ with the formula $\bigwedge_{p\in\mathbb{P}_s[t]} p$, and every occurrence of a projection term p in φ with the variable term $path(p)$.

Example 5.5 Given custom sorts $\mathsf{Request}\{\mathsf{url} : \mathsf{Str}, \mathsf{user} : \mathsf{User}\}$, $\mathsf{Report}\{\mathsf{reason} : \mathsf{Str}, \mathsf{user} : \mathsf{User}\}$ and $\mathsf{User}\{\mathsf{name} : \mathsf{Str}\}$, the formula
$\mathsf{Request}(r) \Rightarrow \Diamond_I \exists u.\ \mathsf{Report}(u) \wedge r.user = u.user$ is compiled to:

$\exists r\_url, r\_user, r\_user\_name.\ \mathsf{Request}(r, r\_url, r\_user) \Rightarrow$
$\Diamond_I \exists u, u\_reason, u\_user.\ \mathsf{Reason}(u, u\_reason, u\_user) \wedge$
$\mathsf{User}(u\_user, u\_user\_name) \wedge r\_user\_name = u\_user\_name.$

Compilation of aggregations We currently do not allow grouping by or aggregating over terms of complex sorts. Hence, aggregations do not need to be compiled. To group by terms of complex sorts, users may assign individual fields of primitive sorts to distinct variables instead, as shown in Example 5.6.

Example 5.6 Assuming the custom sorts from example 5.5 on page 26, we want to count the number of requests per user. To do so, we group requests by their users' properties and count the occurrences of request events per user: $\mathsf{c} \leftarrow \mathsf{CNT\ i;\ name}\ (\blacklozenge_{[0,*]}\, \mathsf{Request(r)} \wedge \mathsf{name} = \mathsf{r.user.name} \wedge \mathsf{tp(i)})$.

Compilation of booleans and null Unlike CMFODL, MFODL has no support for boolean-valued terms or the constant null. We instead regard booleans in MFODL as integers and compile the constant term `false` to the integer value 0 and `true` to the value 1: $comp_{bool} : \mathsf{B} \to \mathsf{Z} := c \mapsto 0$ if $c = \mathsf{False}$, else 1. Based on the derivation rules for terms in Figure 4.1, the only operation applicable on terms of sort B is equality (rule EQ). And indeed, $\forall c_1, c_2 \in \mathsf{B}.\ c_1 = c_2 \leftrightarrow comp_{bool}(c_1) = comp_{bool}(c_2)$. Similarly, we compile every null constant to the integer 0 and apply the same proof idea for correctness.

Chapter 6

Monitorability

This chapter elaborates on monitorable formulas and lifting the monitorability property from MFODL to CMFODL.

As mentioned in Section 2.3, MonPoly only supports monitoring a syntactic fragment of MFODL formulas. The formulas in the fragment are referred to as *safe*. Therefore, MonPoly needs to initially verify the monitorability of a given input formula. Besides validating the well-formedness of a formula, as presented in Chapter 4, and ensuring that all future intervals of temporal operators are finitely bounded, the monitorability check must verify that all intermediate results are finite when evaluating the input formula bottom-up. In other words, all relation operations corresponding to the semantics of the involved subformulas must result in finite relations.

Technically, the existing monitorability checks for MFODL formulas can still be run on formulas compiled from CMFODL formulas. By doing so, the user-facing error messages are related to the compiled formula instead of the original CMFODL formula. To improve the user experience, we lift the monitorability rules to CMFODL formulas as a stricter approximation of the existing rules for MFODL formulas. This way, whenever a compiled CMFODL formula is considered safe, the monitoring process continues independently of the monitorability verdict on the CMFODL formula. Otherwise, the monitorability check on the CMFODL input formula is also guaranteed to fais, providing improved user feedback.

To implement a stricter approximation of the monitorability rules for CMFODL formulas, we generalize the function $\mathsf{tvars} : \mathsf{term} \rightarrow \mathsf{var\ list}$, that returns the set of free variables for a given term, with the function ctvars: for every free variable of some complex sort, all leaves of the variable are considered to be free variables too.

Proposition 6.1 *Given a function* $\mathsf{m} : (\mathsf{term} \rightarrow \mathsf{var\ list}) \rightarrow \mathsf{formula} \rightarrow \mathsf{bool}$*, which returns a monitorability verdict for a given formula and a function*

returning the free variables of a term, $\mathsf{m}(\mathsf{ctvars},\mathsf{f}) \implies \mathsf{m}(\mathsf{tvars},\mathsf{f}')$ *is satisfied for any formula* f *and its compiled counterpart* f'*.*

In this work, we only provide a proof idea for Proposition 6.1: we can prove it by structural induction over the structure of formulas. Base cases cover all non-recursive variants, such as binary arithmetic and equality, where we show that each case satisfies Proposition 6.1. Subsequently, we assume Proposition 6.1 to hold for any formula f, and show that this implies the satisfaction of the proposition for each recursive formula on f.

One disadvantage of our approach is that the monitorability check on a complex-typed formula might fail for a different reason than the monitorability check of the compiled formula. Because we only output the first of possibly many issues, the user experience may degrade in these cases. This effect is inherited from the existing monitorability check on MFODL formulas, which only reports on the first issue, disregarding the total number of errors.

Chapter 7

Extending MonPoly

This chapter gives an overview of the MonPoly extensions providing support for monitoring complex data types. We first present the extended grammar of signatures, formulas, and log files accepted as valid inputs by MonPoly. Next, we describe the internal monitoring pipeline and the implementation of specific algorithms described in previous chapters. All extensions mentioned in this chapter are implicitly backward compatible with inputs for earlier versions of MonPoly.

7.1 Input format

To monitor an application log stream, MonPoly requires additional inputs besides the log stream under monitoring [7]: The formula file contains the property under monitoring, written as a CMFODL formula. Its concrete syntax is explained in Section 7.1.1. The signature file describes all custom sorts used by the input formula. The accepted syntax of signature files is described in Section 7.1.2. At last, 7.1.3 describes the expected format of log streams.

7.1.1 Policy format

Monitoring policies in MonPoly are formulated in a syntax corresponding to complex-typed MFODL introduced in Chapter 3. Figure 7.1 describes the concrete syntax of a well-formed formula file. The functions `f2i`, `i2f`, `f2s`, `s2f`, `i2s`, `s2i` may be used to convert terms between floats, integers and strings. Table 7.1 maps the mathematical notation of MFODL formulas to their corresponding notation in the concrete formula syntax.

7.1.2 Signatures format

Signature files describe the signature $\Delta = (\mathsf{PS}, S_{def}, \{\})$ used by a formula [7]. Specifically, signature files allow users to declare all custom sorts $s \in S_{def}$.

```
formula  ::=
         | TRUE | FALSE
         | (, formula , ) | NOT, formula
         | term , (= | > | < | <= | <=), term
         | formula , (EQUIV | IMPLIES | AND | OR), formula
         | (EXISTS | FORALL), var-list , . , formula
         | var , <-, aggreg , var , [ ; , var-list ], formula
         | unop_i , [ interval ], formula
         | formula , (SINCE | UNTIL), [ interval ], formula
         | term , SUBSTRING, term
         | term , MATCHES, term , [(, (_ | term ), {, (_ | term )})]
         | pred
         | (|> | MATCHF | FORWARD), [ interval ], fregex
         | (<| | MATCHP | BACKWARD), [ interval ], pregex
         | (LET | LETPAST), pred , IN, formula
pred     ::= ident , (, [ term , {, , term }], )
unop_i   ::= NEXT | PREV | EVENTUALLY | ONCE | ALWAYS | PAST_ALWAYS
aggreg   ::= CNT | SUM | AVG | MED | MIN | MAX
interval ::= (( | [), bound , , , ( bound | *), () | ])
bound    ::= integer , [s | m | h | d]
var-list ::= var , {, , var }
term     ::=
         | term , (+ | - | * | / | MOD), term
         | -, term | (, term , )
         | (f2i | i2f | i2s | s2i | f2s | s2f | r2s | s2r), (, term , )
         | (DAY_OF_MONTH | MONTH | YEAR | FORMAT_DATE), (, term , )
         | var | cst | term , . , ident
fregex   ::=
         | (, fregex , ) | . | fregex , formula , ?, | fregex , fregex
         | fregex , +, fregex | fregex , *, fregex
pregex   ::=
         | (, pregex , ) | . | pregex , formula , ?, | pregex , pregex
         | pregex , +, pregex | pregex , *, pregex
cst      ::=
         | integer | rational | ", string , " | true | false
         | ident , {, { ident , :, cst }, }
var      ::= ident
ident    ::= ( letter | digit | _), { letter | digit | _}
```

Figure 7.1: EBNF-form of well-formed formula files

symbol	MonPoly terminal	assoc.
$\neg$	`NOT`	none
$\wedge$	`AND`	left
$\vee$	`OR`	left
$\rightarrow$	`IMPLIES`	right
$\leftrightarrow$	`EQUIV`	left
$\exists\ \forall$	`EXISTS FORALL`	none
S U	`SINCE UNTIL`	none
$\bullet\ \circ\ \blacklozenge\ \lozenge\ \blacksquare\ \square$	`PREV NEXT ONCE EVENTUALLY PAST_ALWAYS ALWAYS`	right

Table 7.1: Mapping between mathematical and MonPoly notation

The predicate-based signatures used by earlier versions of MonPoly are still supported. Figure 7.2 shows the syntax of extended signature files. What now follows is a set of rules and definitions related to declaring well-formed signatures:

Definition 7.1 (Top-level sorts) We define the set of custom sorts declared in a signature file and prefixed with the keyword event as TopLevel. For every time point i, $\{s \mid (\_,s) \in \mathbb{O}\} \subseteq \mathsf{TopLevel}$ is satisfied.

Definition 7.2 (Inline sorts) The syntax declared in Figure 7.2 allows the declaration of inline product sorts. This syntactic feature improves the readability of deeply nested data structures by inlining the declaration of a nested sort. Inline sorts are automatically extracted to separate product sort definitions during parsing and have no further semantic interpretation.

Example 7.3 Using inline sorts, we can describe a custom sort Request as $\mathsf{Request}\ \{\mathsf{url} : \mathsf{string},\ \mathsf{user} : \{\mathsf{name} : \mathsf{string}\}\}$. This signature is equivalent to $\mathsf{Request}\ \{\mathsf{url} : \mathsf{string},\ \mathsf{user} : \mathsf{Request\_user}\};\ \mathsf{Request\_user}\ \{\mathsf{name} : \mathsf{string}\}$.

Rule 7.4 (No recursive sorts) As declared in Section 3.1, there exists a well-formedness constraint prohibiting recursion between sorts. Therefore, the sort of a declared product sort field must not reference its parent sort directly or indirectly.

Rule 7.5 (Unique top-level sorts) Top-level sorts as defined in Definition 7.1 must be pairwise structurally distinct. This is required by the sort matching algorithm introduced in Section 7.2.2.

As described in Figure 7.2, the type null can only be used to declare the sort of a field because it is a JSON-specific data type. It is still helpful to pattern-match against nullable fields, as shown in Example 7.6.

Example 7.6 Assume a JSON log stream of events, where each event has a type and an optional occurrence count, declared either as an integer value or

```
signature        ::= { product-sort | predicate-symbol }
predicate-symbol ::= ident , (, pred-arg , { ,, pred-arg }, )
pred-arg         ::= [ ident :], primitive-sort
primitive-sort   ::= string | int | float | bool
product-sort     ::= [event], ident , product-body
product-body     ::= {, product-field , { ,, product-field }}
product-field    ::= ident : field-type
field-type       ::= primitive-sort | ident | record-body | null
ident            ::= ( letter | digit | _), { letter | digit | _}
```

Figure 7.2: EBNF-form of well-formed signature files

by the default value null. We want to aggregate the sum of the number of occurrences, grouped by the event type, where null counts as 1. The following listings present a signature and formula of a possible solution, making use of the null type as field sort:

```
1 event Event {
2   type: string,
3   count: int
4 }
5 event DefaultEvent {
6   type: string,
7   count: null
8 }
9
```

Listing 7.1: Signature describing variants of events

```
1 s <- SUM count; type (ONCE [0,*]
2   (EXISTS e. (Event(e) AND type = e.type AND count = e.count)) OR
3   (EXISTS e. ( DefaultEvent (e) AND type = e.type AND count = 1))
4 )
5
```

Listing 7.2: A formula aggregating over variants of events

7.1.3 JSON log format

One of the main benefits of this work is the possibility of monitoring JSON-based logs without fundamental transformations. Nevertheless, the newly introduced JSON log parser of MonPoly only supports a subset of log file formats. Despite the introduction of standards for event-driven application logs, such as XES [19] or XOC [21], applications may use their own custom file formats to output streams of logged events. To accommodate this fact, the JSON log parser of MonPoly keeps the number of requirements regarding

```
log           ::= { time-point | command }, [ ts ]
time-point    ::= ts , json-record , { newline , json-record }
command       ::= >, command-name , [ string { ,, string }], <
command-name  ::=
                | print | terminate | print_and_exit | get_pos
                | save_state | save_and_exit | set_slicer
                | split_save
ts            ::= @, integer
```

Figure 7.3: EBNF-form of the supported JSON log format

the format of the log input as small as possible to allow monitoring large sets of application logs with minimal transformation.

Figure 7.3 describes the supported log stream format. Every top-level JSON structure in the input log stream must be a JSON record value, while other JSON document types are not supported yet. Every top-level JSON record must either directly follow a timestamp token or appear on a separate line. Each timestamp token declares the beginning of a new time point and defines the assigned timestamp on all JSON records occurring after it. Finally, the newly introduced log parser maintains support for MonPoly commands.

7.2 Monitoring pipeline

This chapter describes the internal data flow of a monitoring process between the components of MonPoly. Figure 7.4 provides an overview of all involved components. While solid lines represent the single passing of data, dashed lines describe data streams. `*.mfotl`, `*.sig` and `*.log` describe input files for formulas, signatures and log streams. We further introduce some declarations related to JSON: The set JsonValue describes the set of all well-formed JSON documents of type object, array, string, number, boolean and null [22]. The set $\mathsf{JsonObject} := \mathcal{P}(\mathsf{Str} \times \mathsf{JsonValue}) \subset \mathsf{JsonValue}$ consists of all JSON object values, where each element is represented as a set of tuples mapping a field name to a JSON value.

Steps 1 and 2 represent the parsing and type checking of a CMFODL formula. Section 7.2.1 describes the data structure `cplx_formula` in more detail. The type checker, implementing the algorithms described in Chapter 4, annotates the parsed formula with the derived type information, stored in a data structure `tctxt`: It consists of the symbol table, predicate schema, and custom sorts of the current formula. The type-checked formula is then forwarded to the preprocessing and compilation (Steps 3 and 4), described in Chapter 5. Before the compiled formula is passed to the MonPoly monitor, its moni-

torability properties from Chapter 6 are validated in Step 5. If the validation fails, the monitoring process is halted.

Based on the presence of the command line flag `--json`, the content of the log stream is either passed to the existing parser of MonPoly or the newly implemented JSON log parser (Steps 8 and 9). In the case of a JSON log stream, the parsed JSON objects are matched against custom signatures in Step 10. This process is explained in Section 7.2.2. Step 7: *Create Relation Schema* represents the implementation of Algorithm 2, described in Section 5.1. Similarly, the tuples output by Steps 9 and 10 and generated based on Section 5.2, describing the transformation of complex-typed temporal structures.

Finally, the compiled formula, the relation schema, and the stream of tuples from the log input are passed to MonPoly 's monitoring component in Step 11.

7.2.1 Formula annotations

Type information of formula terms is not only relevant during type checking of an input formula, but also useful for succeeding tasks such as compilation and monitorability checks. MFODL constructs such as quantifiers, let statements, and aggregations introduce nested scopes, where variables from outer scopes may be shadowed. To retain the type information of variables in nested scopes, a single global symbol table and predicate schema will not suffice. We instead extend the data structure describing a formula with an additional field storing some polymorphic formula annotation: `type 'a cplx_formula = ('a * formula_ast)`, where `formula_ast` references the data structure described in Figure 7.1. This allows the type checker to store formula- and scope-specific type information, which subsequent formula transformations can access.

7.2.2 Matching custom sorts

Chapter 3 introduces the sets of domain values $\mathbb{D}$ and objects $\mathbb{O} \subseteq \mathbb{D}$, where each element is tagged with its corresponding sort. On the other hand, a JSON object read from a log entry is not tagged. To derive the custom product sort of a JSON object, we need to match the structure of the JSON object against every sort $s \in \mathsf{TopLevel}$ declared in the signature. We require the partial mapping from JSON objects to sorts to be well defined. Therefore, all sorts $s \in \mathsf{TopLevel}$ must be pairwise structurally distinct. We further allow the mapping from JSON objects to signatures to be partial: Whenever a JSON object is encountered whose structure does not match that of any custom sort, a warning is printed to the application output, but no error is raised. This allows the monitoring of application logs where the set of all custom sorts $\{s \mid (\_, s) \in \mathbb{D}\}$ is not known in advance. Finally, all JSON fields of list types are ignored when matching their structure. This allows the monitoring of log

streams containing list structures without the explicit support of describing them as part of a custom sort.

Algorithm 4 describes the mapping as a recursive function: For each field of a given JSON object, we distinguish three cases: If the field's value is of a list type, we ignore it and continue. Whenever the field's value is a nested object, we call `find_sort` recursively, only matching against the sort of the corresponding field. In any other case, we compare the type of the field value with the corresponding sort using the relation operator $\approx := \{(\mathsf{string}, \mathsf{Str}), (\mathsf{int}, \mathsf{Z}), (\mathsf{float}, \mathsf{F}), (\mathsf{boolean}, \mathsf{B}), (\mathsf{null}, \mathsf{Null})\}$. For a given JSON object, `find_sort` is then initially called on line 23 to match against all custom sorts in TopLevel.

Algorithm 4: Match JSON objects against custom sorts

```
   Data: JSON object o ∈ JsonObject
 1 Function find_sort(S) is
       Data: Set of custom sorts S ⊆ S_def.
       Result: Custom sort s ∈ S of o, or ⊥.
 2     foreach sort s ∈ S do
 3         foreach field f ∈ o do
 4             τ_json ← typeof o[f];
               /* ignore fields of list types:                    */
 5             if τ_json = array then
 6                 continue
 7             end
 8             if τ_json = object then
                   /* recursively match sort:                     */
 9                 if find_sort(o[f], {(s', _) ∈ S_def | s' = s(f)}) = ⊥ then
10                     return ⊥;
11                 else
12                     continue
13                 end
14             end
15             τ_sort ← s(f);
16             if τ_json ≉ s(f) then
17                 return ⊥;
18             end
19         end
           /* All fields have matched:                            */
20         return s;
21     end
22 end
23 call find_sort({(s, _) ∈ S_def | s ∈ TopLevel});
```

After deriving the custom sort of a JSON object, the object is transformed to a set of tuples, as described by Algorithm 3 in Section 5.2. Similarly to the compilation of constant boolean values in MFODL formulas explained in Section 5.3.2, JSON values `true` and `false` and registered as integer values `1` and `0`.

The current implementation of the JSON log parser generates object identifiers by assigning an incrementing counter to each tuple on registration. The parser resets the counter at the beginning of a new time point. This guarantees that every tuple of the same time point has a unique identifier, enabling unique references between tuples of the same time point. On the other hand, local uniqueness implies that unrelated tuples registered at different time points may share the same identifier, as mentioned previously in Section 5.2.

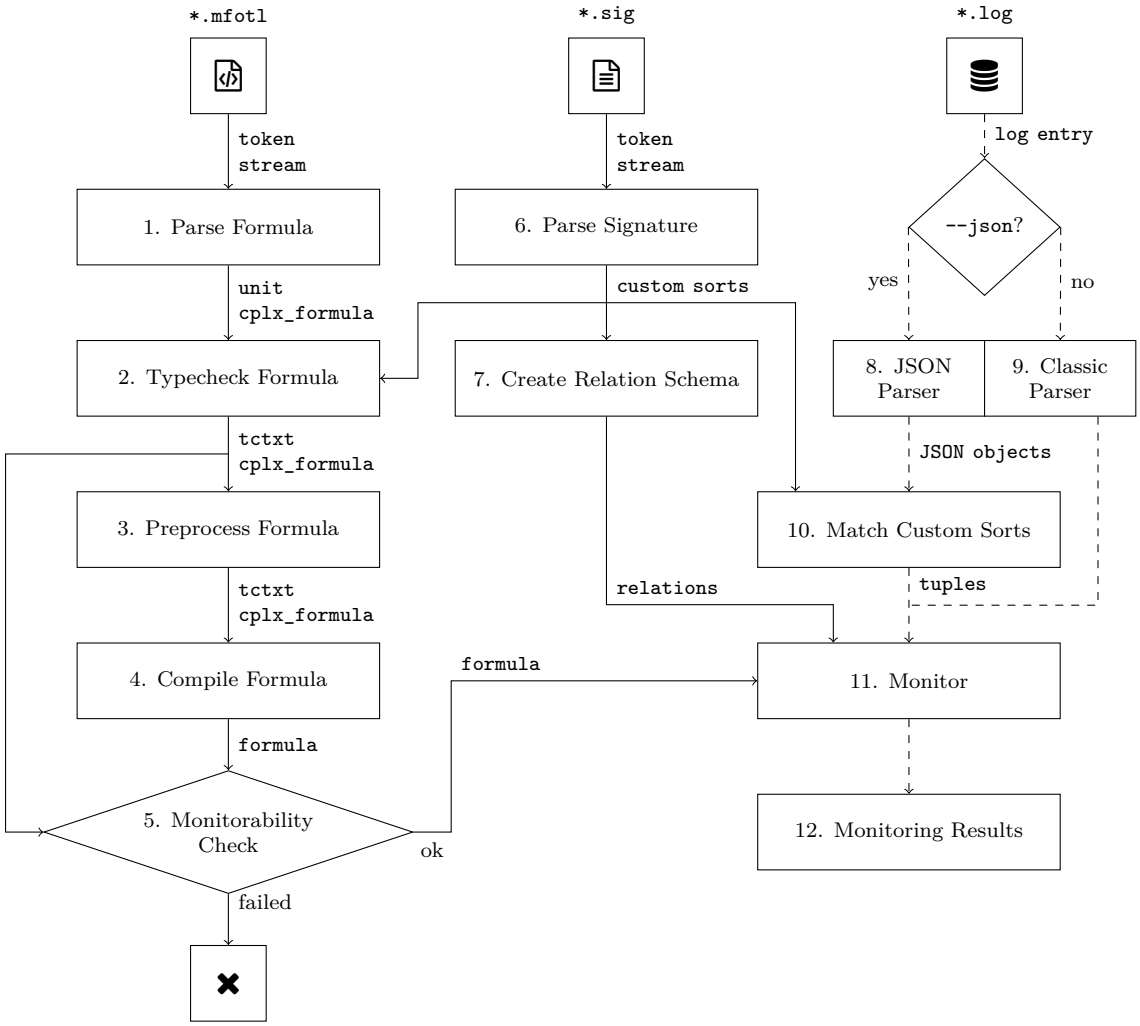

Figure 7.4: Data flow of the monitoring pipeline

Chapter 8

Case Study

In the following two sections, we present both a generalized and a specialized preprocessor for monitoring complex-typed event streams with MFODL and MonPoly. Both cases are related to Definity. They use MFOTL and MonPoly to verify policies against JSON-formatted application logs collected during system tests [18]. We then show how CMFODL can be used to simplify preprocessing and improve the readability and maintainability of the formulas.

8.1 Generalized preprocessing

Using this approach, we monitor any MFODL formulas against arbitrary JSON logs without adapting the preprocessing and the signatures to the particular use case. To achieve this, the generalized preprocessor transforms a stream of JSON objects to a temporal structure over the signature described in Listing 8.1. For every encountered JSON document at time point i in the input log, the preprocessor proceeds as follows: For every (possibly nested) JSON object, it generates a unique identifier id and outputs the tuple (id) on the relation root. It then continues by traversing the fields of the object: for each field named f of type int, float, string, bool or null, a new identifier id_{val} is generated for the field value v and written as the tuple (id_{val}, v) (or (id_{val}) for constant values true, false and null) to the corresponding relations int, float, string, true, false or null. Finally, the tuple (id, f, id_{val}) is written to the relation key, introducing a reference from the value to its object under the field f. In the case of a field value being an object, the procedure is repeated recursively. JSON arrays are processed similarly to objects: Instead of writing a tuple to the key relation, we generate a tuple $(id, index, id_{val})$ in the relation idx, establishing a reference between the array with identifier id and the value with identifier id_{val} of the element at index $index$. To generate the correct timestamp for each time point, the preprocessor accepts a JSON path to the (possibly nested) field containing the timestamp of each JSON log entry.

When providing the JSON log described in Listing 8.2 as input to the preprocessor, together with the timestamp path event.log_entry.time, the transformed log in Listing 8.3 is produced.

We formulate the following specification in MFODL over the representation of the temporal structure in Listing 8.3: *"Refute every program trace containing an event of type* Log *where* event.log_entry.level *is equal to* ERROR *or* CRITICAL*."* Listing 8.4 presents a possible MFODL formula for this specification.

We now solve the same task by formulating the specification in CMFODL instead and applying the extensions of MonPoly presented in this thesis. We additionally make use of a minimal log preprocessor described in Appendix A.1 that extracts the timestamp of each log entry, such that the input log stream to MonPoly conforms to the grammar described in Listing 7.3. The CMFODL signature is described in Listing 8.5, while the CMFODL formula is shown in Listing 8.6. First, the CMFODL signature describes the structure of the input log exactly, making it easier for users to understand the involved data structures. Comparing the MFODL formulas in Listing 8.4 with the CMFODL formula in Figure 8.6 shows that the same specification can be formulated much more naturally in CMFODL and therefore reduces the risk of introducing subtle bugs while formulating specifications.

```
root(id:int)                                    (* object          *)
int(id:int, value:int)                          (* int value       *)
float(id:int, value:float)                      (* float value     *)
str(id:int, value:string)                       (* string value    *)
true(id:int)                                    (* constant true   *)
false(id:int)                                   (* constant false  *)
null(id:int)                                    (* constant null   *)
key(object:int, field:string, value:int)        (* object field    *)
idx(array:int,index:int,value:int)              (* array element   *)
```

Listing 8.1: Generalized MFODL signature

```
{
  'type':"Log",
  'event':{
    'src':{'a':0},
    'log_entry':{
      'level':"INFO",'time':1648053358,
      'message':"Configuration upated",'module':"auth",'line':17,
      'host':"127.0.0.1"
    }
  }
}
{
  'type':"Log",
  'event':{
    'src':{'a':0},
    'log_entry':{
      'level':"ERROR",'time':1648053380,
      'message':"Power loss",'module':"power",'line':567,
      'host':"127.0.0.1"
    }
  }
}
```

Listing 8.2: Input JSON log

```
@1648053358 root(0); key(0,"type",1)(3,"a",4)(2,"src",3)(5,"level",6)
   (5,"time",7)(5,"message",8)(5,"module",9)(5,"line",10)(5,"host"
   ,11)(2,"log_entry",5)(0,"event",2); int(4,0)(7,1648053358)
   (10,234); str(1,"Log")(6,"INFO")(8,"Configuration updated")(9,"
   auth")(11,"127.0.0.1")
@1648053380 root(12); key(12,"type",13)(15,"a",16)(14,"src",15)(17,"
   level",18)(17,"time",19)(17,"message",20)(17,"module",21)(17,"
   line",22)(17,"host",23)(14,"log_entry",17)(12,"event",14); int(
   16,0)(19,1648053380)(22,567); str(13,"Log")(18,"ERROR")(20,"Power
    loss")(21,"power")(23,"127.0.0.1")
```

Listing 8.3: Output of the generalized preprocessor

```
LET Log(level, message, host) = EXISTS r. root(r) AND
  (EXISTS t. key(r, "type", t) AND str(t, "Log")) AND
  (EXISTS e, le, lvl, msg, id. key(r, "event", e) AND
    key(e, "log_entry", le) AND
    key(le, "level", lvl) AND str(lvl, level) AND
    key(le, "message", msg) AND str(msg, message) AND
    key(le, "host", id) AND str(id, host))
IN
Log(level, message, host) IMPLIES
  (NOT level = "CRITICAL" AND NOT level = "ERROR")

```

Listing 8.4: MFODL formula

```
LogEvent {
  src: {a: int},
  log_entry: {
    level: string,
    time: int,
    message: string,
    module: string,
    line: int,
    host: string
  }
}

event Log {
  type: string, event: LogEvent
}
```

Listing 8.5: CMFODL signature

```
LET is_error(event) =
  event.log_entry.level = "ERROR" OR
  event.log_entry.level = "CRITICAL"
IN
Log(l) AND l.type = "Log" IMPLIES NOT is_error(l.event)
```

Listing 8.6: CMFODL formula

8.2 Specialized preprocessing

We now look at the current approach of DFINITY, which utilizes a specialized preprocessor [18]. In this case, we do not intend to entirely replace the log preprocessing with CMFOTL since the preprocessing is needed to enrich the application logs with additional information required to verify the desired properties. Instead, we want to show that even in cases where log preprocessing is necessary, CMFODL still provides a benefit by allowing users to formulate specifications so that the resulting formulas naturally correspond to the underlying structure of the event data. This increases the readability and maintainability of formulas.

To clarify this argument, we look at the abstract application event *reboot*, referred to in multiple policies including reboot_count [18]. DFINITY's current PYTHON-based preprocessor generates a tuple $(host\_addr, data\_center\_prefix)$ of the relation reboot whenever the abstract event *reboot* occurs, namely whenever a JSON log entry is encountered with the following properties: The field _source.host.ip is equal to *host_addr*, the field _source.syslog.identifier is equal to the string "systemd", and the message of the log entry under _source.message is equal to the string "Starting IC replica...". In a corresponding MFODL formula, the occurrence of a reboot can therefore be verified by the predicate formula reboot(host_addr, data_center_prefix).

With CMFODL, we achieve the same result by introducing a custom predicate as shown in Listing 8.7. The advantage of the latter approach is that we can formalize the same abstractions of events by using a single language in a single place without the need for synchronization. This improves the readability and comprehensibility of the formulas. While the CMFODL formula in Listing 8.7 may look more complex than the original MFODL formula, it is comparable to the corresponding PYTHON-code of the preprocessor [18]. In addition, the concept of CMFODL modules could be introduced to improve the reusability of policies.

```
1 LET reboot(event, host_addr, data_center_prefix) =
2   event._source.host.ip = host_addr AND
3   event._source.host.data_center_prefix = data_center_prefix AND
4   event._source.syslog.identifier = "systemd" AND
5   event._source.message = "Starting IC replica..."
6 IN
7 Event(e) AND reboot(e, addr, prefix)
```

Listing 8.7: CMFODL formula abstracting a reboot event

Chapter 9

Considerations

In this chapter, we elaborate on different considerations regarding the specification and implementation of complex data types.

9.1 Scope of unique object identifiers

As described in Algorithm 3 of Section 5.2, we assign unique identifiers to tuples generated from complex JSON object values. The current implementation uses a simple integer-based counter, which the monitor resets at the beginning of each time point – therefore assigning only locally unique identifiers. The greatest advantage of this implementation is its simplicity, while it still allows unique referencing between tuples of the same time point. On the other hand, non-global uniqueness implies that unrelated objects of different time points share the same identifier. Therefore, we cannot use the value of an identifier in different temporal scopes (such as in `UNTIL` or `EVENTUALLY`) without risking an unintended change of semantics of the input formula. Furthermore, we cannot define the more efficient referential equality between objects observed at different time points. Instead, we must rely on structural equality by preprocessing complex-typed formulas before compilation. The example in Section 5.3.1 illuminates possible issues when dealing with locally unique identifiers.

An alternative implementation may depend on globally unique identifiers instead. Assigning every observed object a unique identifier may not be helpful: every referential equality between two objects would evaluate as false. Instead, objects with equal structure should be assigned the same identifier. Referring back to the example of Section 5.3.1, every object instance related to the same user would share the same identifier, allowing to pass identifiers over temporal operators and enabling referential equality. This would significantly simplify the structure of compiled formulas by skipping the transformations during preprocessing explained in Section 5.3.1. The downside of globally unique identifiers is the complexity of the implementation: Assigning shared

identifiers to structurally equal objects at different time points requires the monitoring system to keep track of a mapping from object structures to their assigned identifiers. To limit the space complexity of this map, obsolete entries must be garbage collected regularly, based on the temporal reach of the formula under monitoring.

To keep the scope of this work limited, we have chosen an implementation with locally unique identifiers instead.

9.2 Orderable custom sorts and boolean sorts

As declared by the type derivation rules in Figure 4.2, values of custom product and boolean sorts are not (totally) ordered. For both kinds of sorts, the ordering of values may be interpreted differently based on the application. Introducing an order in this work and changing it later would not guarantee backward compatibility with formulas written for older versions of MonPoly. Moreover, authors of formulas may always declare a particular order relation on custom sorts using let bindings. For these reasons, we do not introduce a fixed order for custom and boolean sorts.

9.3 Reporting values of nested fields

Whenever MonPoly encounters a prefix of a trace satisfying a given MFODL formula, it prints the corresponding interpretation of the formula, i.e. the values assigned to all free variables at the evaluated time point. Whenever a free variable is of a complex sort, the variable's value points to the compound value's identifier, as described in Chapter 5. While the compilation process does generate new variables for each projection in the original CMFODL formula, each of these variables is bound by an existential quantifier and is therefore not a free variable of the input formula. A possible solution is to avoid binding variables representing nested fields of free variables. This approach changes the set of free variables during compilation as a side effect. We instead delegate the decision on which values to report to the author of the formula, by allowing assignments of nested values to arbitrary free variables. Referring to Example 5.5 on page 26, if the URL of a violating request should be reported, one can extend the formula in conjunction with the assignment $\mathsf{url} = \mathsf{r.url}$, introducing a new free variable url.

Chapter 10

Conclusion

MonPoly and MFODL provide a broad set of features, including real-time monitoring, global quantification, regular expressions over program traces, and SQL-like aggregation operators. While JSON-formatted application logs have become ubiquitous, allowing systems to output events as arbitrary complex data structures, both MonPoly and MFODL lack support for custom event data types.

In this work, we extended MonPoly 's specification language MFODL to support compound domain values of custom product sorts. Most importantly, the extension is backward compatible, so all MFODL formulas are valid CMFODL formulas. The extension allows projection on nested fields and adds support for Boolean constants. Therefore, CMFODL can be used to formulate specifications on parametrized program traces, where the domain of supported event data covers all possible JSON documents, except for lists. Combined with transforming temporal structures and compiling of CMFODL to MFODL formulas implemented as an extension to MonPoly , this allows users to monitor arbitrary JSON logs without or only minimal preprocessing involved. By avoiding complex log preprocessing, an additional possible source of errors in the monitoring pipeline can be avoided. Finally, the extension to MonPoly is backward compatible with signature files and log files written for earlier versions of MonPoly.

In conclusion, the fact that we can compile CMFODL formulas and complex-typed temporal structures such that MonPoly can monitor them implies that CMFODL is generally not more expressive than MFODL. Instead, it offers a more comprehensible interface for users to formulate specifications over JSON logs, therefore reducing the risk of introducing errors in the specifications.

10.1 Future work

This section discusses open questions and extensions of the work presented in this thesis.

10.1.1 Temporal invariant object references

A large part of the current compilation process of CMFODL formulas, namely the preprocessing, is only required because identifiers assigned to objects are only locally unique. Section 9.1 compares this approach with other strategies, namely globally unique identifiers. More generally, making references between objects invariant to time would allow us to simplify the compilation process and the complexity of output formulas. Future projects may research suitable data structures for storing mappings between object identifiers and object instances that can be garbage-collected based on the temporal scope of an input formula.

10.1.2 Sum Types

Example 7.6 on page 32 shows how we can make use of the sort match algorithm described in Section 7.2.2 to pattern-match against field values of different sorts. Nevertheless, this approach is limited to top-level sorts and may decrease the readability of signatures as soon as deeply nested data structures are involved. The concept of sum types may improve on this by allowing users to declare a custom sort as an untagged or tagged (disjoined) union. As an example, one could declare the sort of a field as string | null (untagged) or Ok of Result | Err of Error for some custom sorts Result, Error (tagged). This extension would need to be accompanied by a syntax extension of CMFODL, allowing users to match against different variants in formulas to extract the actual value during runtime. Possible challenges may arise for matching untagged JSON values against variants of sum types and the compilation to a safe MFODL formula, such that it is monitorable by the current implementation of MonPoly.

10.1.3 Support for list sorts

The extension of MFODL and MonPoly presented in this thesis does not support list-like sorts. Therefore, JSON arrays part of an input log file are ignored during parsing and cannot be accessed in a CMFODL formula. Representing elements of lists as tuples of relations can be solved similarly to the solution presented for product sorts in Section 5.2, namely by normalizing the relations. In the case of lists, elements of a list can be stored in a separate relation, where each tuple represents an element and contains an additional reference pointing to the tuple owning the list. Exposing the content of a

list in formulas might be more challenging. Local quantification over the domain of elements in a list, as introduced by ParTraP [9] and others, would be useful but requires significant changes to the underlying monitor system. Another approach may provide a generalized fold function over the domain of a list. Eventually, lists of unbounded size introduce separate challenges to normalizing relations.

Chapter 11

Related Work

This chapter looks at other approaches with expressive specification languages for runtime verification. We focus on languages that provide support for complex-structured event data, as they are pursuing a similar goal as CMFODL.

11.1 LOGSCOPE

LogScope is a runtime verification tool initially developed for NASA's Jet Propulsion Laboratory [2]. It provides a temporal specification language that is translated into an automata-based monitor. Specifications can be formulated in either temporal logic formulas or a parameterized ∀-automaton that supports event parameters and universal quantification. Constraints on events, called event predicates, can be specified in general Python expressions over arbitrary data structures carried by the events. Furthermore, LogScope allows formulating custom side effects in Python code, called event actions, triggered on the successful evaluation of an event predicate. Side effects may manage global state referenced in event predicates. LogScope may therefore be regarded as a mix of declarative and operational language. However, compared to CMFODL, neither LogScope's temporal logic pattern, nor its automaton language allows referencing data of past events in an event predicate. Certain properties formulated in CMFODL, such as $\mathsf{A}(\mathsf{x}) \rightarrow \blacklozenge_{[0,*]}\,(\mathsf{B}(\mathsf{y}) \wedge \mathsf{y} = \mathsf{x})$, can therefore not be represented in LogScope. Furthermore, side effects may introduce additional complexity to a specification, reducing its comprehensibility. Finally, the LogScope framework is not suitable for online runtime verification: a program trace is consumed as a Python list of arbitrary event objects and can not be streamed in real-time.

11.2 LTL-FO$^+$

LTL-FO$^+$ [14] is the runtime verification counterpart to CTL-FO$^+$ [15] developed for model checking. LTL-FO$^+$ supports online runtime monitoring over program traces consisting of arbitrary XML-based messages. Where specifications formulated in CTL-FO$^+$ describe properties of all possible execution paths, specifications written in LTL-FO$^+$ focus on a single program trace. A program trace is modeled as a stream of messages, each representing a relation from parameters to values, where values can be of arbitrarily nested data structures. The specification language LTL-FO$^+$ is an extension of LTL [8]. Compared to CMFODL, LTL-FO$^+$ introduces quantifiers over parameters in a message, but it lacks support for global quantification over an infinite domain of the variables in a program trace. Similarly to LogScope, it provides no support for referencing past events in a specification, which is CMFODL supports.

11.3 PARTRAP

ParTraP has been developed for the runtime verification of medical devices [9]. In this context, a trace consists of inputs captured by device sensors and their interaction with a surgeon. One of the core objectives of ParTraP is to enable software engineers without training in formal methods to formulate trace properties and make them readable by domain experts simultaneously. In ParTraP a program trace is seen as a sequence of events, each represented by an arbitrary nested JSON object. Each event must at least carry its event type and time in a hardcoded format. In contrast, event data is not required to be tagged with their event type when monitored by MonPoly. ParTraP's atomic building blocks are unary *scopes*, such as before A or after first B for some events A, B. By nesting scopes, higher-arity scopes can be constructed. Because events may carry list structures as event data, ParTraP provides *local* quantification over values in lists. For instance, the specification forall a in L, P requires the property P to be satisfied for all values a in the finite list L. However, compared to CMFODL, support for *global* quantification over the infinite domain of the variables on a whole trace is missing.

11.4 HLOLA

Unlike MonPoly , HLola [12] is a stream runtime verification tool. Stream runtime verification (SRV) can be seen as a generalization of monitoring temporal logic formulas. Instead of generating a single Boolean verdict, or a stream of Boolean verdicts produced by MonPoly , SRV describes the dependencies between output streams (results) and input streams (observations)

[13]. HLola uses LOLA as its core language but adds support for arbitrary data types. Input streams acting as the program trace under monitoring can be provided in JSON or CSV format. HLola allows imports of arbitrary library code written in HASKELL, from which it inherits all available data types to describe the structure of input and output streams. As SRV is a generalization of runtime verification, HLola allows the implementation of general LTL operators, including past operators.

Appendix A

Utilities

A.1 Minimal JSON log preprocessor

```sh
#!/usr/bin/env sh

# Usage: extract-ts.sh <TIMESTAMP_PATH>
#
# Formats the content of a JSON log file to be passed to MonPoly.
# The passed JSON path to the timestamp of each log entry follows
# the syntactic rules described at:
# https://stedolan.github.io/jq/manual/#Basicfilters
#
# Example:
# cat json.log | extract-ts.sh ".path.to.timestamp" | monpoly
#
# Dependencies:
# jq : JSON CLI utility

# exit on error, undefined var, pipefail
set -euo pipefail

json_path="$1"
while read line; do
    ts=$(printf '%s\n' "$line" | jq "$json_path")
    printf '@%s %s\n' "$ts" "$line"
done
```

Listing A.1: minimal JSON log preprocessor shell script

Bibliography

[1] Rajeev Alur, Kousha Etessami, and P. Madhusudan. "A Temporal Logic of Nested Calls and Returns". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 467–481. ISBN: 978-3-540-24730-2. DOI: 10.1007/978-3-540-24730-2_35.

[2] Howard Barringer et al. "Formal Analysis of Log Files". In: *Journal of aerospace computing, information, and communication* 7.11 (2010), pp. 365–390.

[3] Ezio Bartocci et al. "Introduction to Runtime Verification". In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 1–33. ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_1. URL: https://doi.org/10.1007/978-3-319-75632-5_1 (visited on 08/23/2022).

[4] David Basin et al. "A Formally Verified, Optimized Monitor for Metric First-Order Dynamic Logic". In: *International Joint Conference on Automated Reasoning*. Springer, 2020, pp. 432–453.

[5] David Basin et al. "Monitoring Metric First-Order Temporal Properties". In: *Journal of the ACM (JACM)* 62.2 (2015), pp. 1–45.

[6] David Basin et al. "Monitoring of Temporal First-Order Properties with Aggregations". In: *Formal methods in system design* 46.3 (2015), pp. 262–285.

[7] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. "The MonPoly Monitoring Tool." In: *RV-CuBES* 3 (2017), pp. 19–28.

[8] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Transactions on Software Engineering and Methodology* 20.4 (Sept. 1, 2011), 14:1–14:64. ISSN: 1049-331X. DOI: 10.1145/2000799.2000800. URL: https://doi.org/10.1145/2000799.2000800 (visited on 08/25/2022).

[9] Yoann Blein et al. "Extending Specification Patterns for Verification of Parametric Traces". In: *Proceedings of the 6th Conference on Formal Methods in Software Engineering*. FormaliSE '18. New York, NY, USA: Association for Computing Machinery, June 2, 2018, pp. 10–19. ISBN: 978-1-4503-5718-0. DOI: 10.1145/3193992.3193998. URL: https://doi.org/10.1145/3193992.3193998 (visited on 08/23/2022).

[10] Jan Chomicki. "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding". In: *ACM Transactions on Database Systems (TODS)* 20.2 (1995), pp. 149–186.

[11] David Basin Sr dan Krstic and Dmitriy Traytel. "Almost Event-Rate Independent Monitoring of Metric Dynamic Logic". In: ().

[12] Felipe Gorostiaga and César Sánchez. "HLola: A Very Functional Tool for Extensible Stream Runtime Verification". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 349–356. ISBN: 978-3-030-72013-1. DOI: 10.1007/978-3-030-72013-1_18.

[13] Felipe Gorostiaga and César Sánchez. "Striver: Stream Runtime Verification for Real-Time Event-Streams". In: *Runtime Verification*. Ed. by Christian Colombo and Martin Leucker. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 282–298. ISBN: 978-3-030-03769-7. DOI: 10.1007/978-3-030-03769-7_16.

[14] Sylvain Halle and Roger Villemaire. "Runtime Monitoring of Message-Based Workflows with Data". In: *2008 12th International IEEE Enterprise Distributed Object Computing Conference*. 2008 12th International IEEE Enterprise Distributed Object Computing Conference. Sept. 2008, pp. 63–72. DOI: 10.1109/EDOC.2008.32.

[15] Sylvain Halle et al. "Model Checking Data-Aware Workflow Properties with CTL-FO+". In: *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007). Oct. 2007, pp. 267–267. DOI: 10.1109/EDOC.2007.36.

[16] Klaus Havelund and Doron Peled. "Runtime Verification: From Propositional to First-Order Temporal Logic". In: *Runtime Verification*. Ed. by Christian Colombo and Martin Leucker. Lecture Notes in Computer Sci-

ence. Cham: Springer International Publishing, 2018, pp. 90–112. ISBN: 978-3-030-03769-7. DOI: 10.1007/978-3-030-03769-7_7.

[17] Klaus Havelund et al. "Monitoring Events That Carry Data". In: *Lectures on Runtime Verification*. Springer, 2018, pp. 61–102.

[18] *Ic/Policy-Monitoring at Master · Dfinity/Ic*. URL: https://github.com/dfinity/ic/tree/master/policy-monitoring (visited on 08/29/2022).

[19] "IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams". In: *IEEE Std 1849-2016* (Nov. 2016), pp. 1–50. DOI: 10.1109/IEEESTD.2016.7740858.

[20] Martin Leucker and Christian Schallhart. "A Brief Account of Runtime Verification". In: *The Journal of Logic and Algebraic Programming*. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07) 78.5 (May 1, 2009), pp. 293–303. ISSN: 1567-8326. DOI: 10.1016/j.jlap.2008.08.004. URL: https://www.sciencedirect.com/science/article/pii/S1567832608000775 (visited on 08/23/2022).

[21] Guangming Li et al. "Extracting Object-Centric Event Logs to Support Process Mining on Databases". In: *Information Systems in the Big Data Era*. Ed. by Jan Mendling and Haralambos Mouratidis. Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2018, pp. 182–199. ISBN: 978-3-319-92901-9. DOI: 10.1007/978-3-319-92901-9_16.

[22] Felipe Pezoa et al. "Foundations of JSON Schema". In: *Proceedings of the 25th International Conference on World Wide Web*. 2016, pp. 263–273.

[23] Dmitriy Traytel. "3.16 Metric First-Order Dynamic Logic and Beyond". In: *A Shared Challenge in Behavioural Specification* (), p. 72.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

MONITORING COMPLEX DATA TYPES

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Zumsteg	Remo

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 07.09.2022	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.