University of California
Santa Barbara

Early Detection of Business Rule Violations

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Isaac A. Mackey

Committee in charge:

Professor Jianwen Su, Chair
Professor Tevfik Bultan
Professor Daniel Lokshtanov

September 2023

The Dissertation of Isaac A. Mackey is approved.

Professor Tevfik Bultan

Professor Daniel Lokshtanov

Professor Jianwen Su, Committee Chair

September 2023

Early Detection of Business Rule Violations

by

Isaac A. Mackey

Acknowledgements

Thank you to my advisor, Jianwen Su, for his support during my graduate studies. I pursued his mentorship after taking his graduate course in logic, CS 209. His guidance in conducting research, academic writing, and giving presentations was invaluable.

Thank you to my committee members, Tevfik Bultan, Daniel Lokshtanov, and Jianwen, for the feedback and suggestions at my program milestones.

Thank you to my many friends at UCSB, especially Freddy Hopp and Cole Hawkins for their company during the COVID-19 pandemic and Neeraj Kumar, Lucas Bang, Glen Frost, and the Verification Lab for being role models of academic excellence.

Thank you to my grandfather Jack for motivating me to finish everything I start.

Finally, thank you most of all to my parents for their enduring support throughout my life and education.

Curriculum Vitæ

Isaac A. Mackey

Education

2023	Ph.D. in Computer Science, University of California, Santa Barbara
2022	M.S. in Computer Science, University of California, Santa Barbara
2016	B.A. in Computer Science, University of Virginia
2016	B.A. in Physics, University of Virginia

Publications

1. Isaac Mackey and Jianwen Su. 2023. Early detection of temporal constraint violations (invited extension, in revision). Information and Computation, Special issue: "Temporal Representation and Reasoning," (2023).
2. Isaac Mackey and Jianwen Su. 2023. Mapping singly-linked rules to linear temporal logic formulas. Information Systems (2023), 102222.
3. Isaac Mackey, Raghubir Chimni, and Jianwen Su. 2022. Early detection of temporal constraint violations. In 29th International Symposium on Temporal Representation and Reasoning (TIME 2022). Schloss Dagstuhl-Leibniz-Zentrum fur Informatik.
4. Isaac Mackey and Jianwen Su. 2022. A Rule-Based Constraint Language for Event Streams. In 4th International Workshop on the Resurgence of Datalog in Academia and Industry. 145-150.
5. Moriba Jah, Emine Basak Alper Ramaswamy, Jeffrey Stuart, Isaac Mackey, and James Crowley. 2020. Improving Navigation Analysis with OD-D: The Visually Interactive Orbit Determination Dashboard. Jet Propulsion Laboratory, NASA, Pasadena, California. (2020).
6. Isaac Mackey and Jianwen Su. 2019. Mapping Business Rules to LTL Formulas. In: Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28-31, 2019, Proceedings. Springer Lecture Notes in Computer Science, 563-565.
7. Yan Tang, Isaac Mackey, and Jianwen Su. 2018. Querying workflow logs. Information 9, 2 (2018), 25.
8. Ernst L. Leiss and Isaac Mackey. 2018. The worst parallel Hanoi graphs. Theoretical Computer Science, Volume 705 (2018), 1-8.

Professional Experience

- Commissioned as a Second Lieutenant in the United States Marine Corps 2023
- Internship in software development, Naval Information Warfare Center Pacific 2021
- Internship in software development, NASA Jet Propulsion Laboratory 2019

Abstract

Early Detection of Business Rule Violations

by

Isaac A. Mackey

The rise of automated systems and sensor networks in virtually all areas of industry and social life means many technologies produce streams of events rich with information. These technologies demand algorithms for evaluating queries on streams, coordinating systems that communicate with events, and monitoring streams with respect to specified constraints. In monitoring, constraints that define correct behavior, e.g., business goals, legal requirements, resource limitations, or safety and security concerns are specified in a formal language; then, an event stream is analyzed at runtime to determine if the constraints are satisfied or violated. To make monitoring effective, it is important to detect constraint violations at the earliest possible time, which we call the early violation detection problem.

We study early violation detection for a class of constraints called rules that restrict time gaps between events and compare events' data contents. We show that (1) the general problem of early violation detection for an arbitrary set of rules is unsolvable and (2) early violation detection is possible for various subclasses of rules. For (1), we show early violation detection is closely related to the problem of finite satisfiability (whether or not a given set of rules can be satisfied by a finite event stream) and prove that finite satisfiability for a set of rules is undecidable with a reduction from the empty-tape Turing machine halting problem, which implies that early violation detection is unsolvable in general. For (2), we study restricted classes of rules. A recent proof of Kamp's Theorem provides a translation algorithm for "dataless" rules through translation to linear temporal logic, yielding formulas hyper-exponential in the size of the input rule.

We present translation algorithms for two subclasses of dataless rules, improving the output size from hyper-exponential to single- and double-exponential, respectively. For rules with data, we first present a technique that calculates deadlines from time gaps between events, then use deadlines for early violation detection for individual rules. We extend these algorithms to monitor an acyclic set of rules by applying a chase process and satisfiability testing. We also report the performance of an implementation of these algorithms. Finally, we consider acyclic sets of rules with aggregation functions over time windows, combining the chase and satisfiability techniques with an encoding of aggregation functions in Presburger arithmetic.

Contents

Chapter 1

Introduction

Events are unorchestrated and asynchronous messages that carry information in software systems. They can communicate the states of devices, signal updates for ongoing or completed processes, and exchange data with external systems. They are a fundamental component in workflow management systems, cyber-physical operations, Internet-of-Things devices, decision support systems, etc., which are ubiquitous in modern society and are still growing in scale and importance. Event-based systems produce time-ordered sequences of events, i.e., event streams, which demand processing to produce useful information about the underlying systems for human operators. Thus, efficient and effective techniques for processing event streams are an open problem for research and a focus of research communities (e.g., [1]).

Workflow management systems are a class of event-based systems that automate business processes, such as order fulfillment and customer service, by organizing the activities of multiple participants (e.g., people and software services) into a coherent workflow. By enacting workflows, workflow management systems produce event streams that serve as a record of the system's execution. One use of these streams is to identify exceptional situations, such as violations of organizational policies, regulations, service-level agreements, or other constraints. These constraints on workflows are collectively referred to as *(business) rules*. For example, a rule may specify that: *Every purchase order must be approved by a manager before the order is fulfilled*, or *Every customer service request must be resolved within 24 hours*. Failure to comply with these rules can result in financial loss, legal liability, or in the worst case, physical harm.

In the last two decades, static verification (e.g., model checking) has been applied to enforce rule compliance in a variety of domains [2, 3]. In this approach, business processes or enabling software are mapped to a formal model, e.g., a Petri net [4] or finite state machines [5, 6], then all possible behaviors of the model are checked against rules in a formal specification language, e.g., linear temporal logic [7]. Unfortunately, static verification is more difficult to apply when participants are allowed flexible behavior, e.g., a provider or client can initiate processes at arbitrary times. Including more flexibility leads to a massive number of possible enactments, making model checking intractable or undecidable for expressive classes of rules. The infeasibility of preventing rule violations with static verification does not indicate a design flaw, rather it follows from the desired flexibility of target applications.

An alternative approach to static verification is *runtime monitoring*, where a monitoring mechanism, often a finite state machine or an online algorithm, observes the service's execution trace and detects constraint violations incrementally. The construction and implementation of monitoring mechanisms from formal specifications is now a central topic in the field of runtime verification [8–10]. In one approach, a monitor is embedded in application software [11, 12], though this method has disadvantages: changes to rules require re-engineering and the overhead of checking rules may increase the application's time and space usage. An alternative approach is to separate the runtime monitor from its target application. This separation abstracts system-level details out of the rule specification and the monitor, meaning rule changes can be implemented quickly. Also, the monitor can be optimized for the specific task of checking rules, e.g., using specialized resources or parallelization, without affecting the application.

Merely identifying exceptional situations may not be sufficient to ensure good system behavior; it is also valuable to identify them *at the earliest possible time*. This allows the system maximum time to take corrective action, such as removing resources from the offending process. In extreme cases for safety-critical systems, late identification could lead to danger for people or property. To identify the earliest possible violation of an LTL formula, reference [13] shows that each violating trace has a minimal prefix that indicates the violation is inevitable; early violation detection can be done by recognizing these prefixes at runtime using finite state machine monitors [14, 15]. Some of these techniques have been extended to handle data in events, e.g., for constraints in first-order linear temporal logic [16,17]. However, these techniques do not address quantitative time constraints, or combinations of quantitative time constraints with constraints on event data, both of which are common in business rules [18].

In this dissertation, we focus on early violation detection for a variety of classes of rules and streams. We start by developing a framework for event streams that carry data and timestamps from completed activities in workflow enactments. We then define a language for specifying business rules, with key features for specifying quantitative time constraints and conditions on event data (Chapter 2), features (or their combination) that are often missing in existing research on runtime monitoring and compliance for business rules. Then, we describe what it means for an event stream to satisfy or to violate a set of rules. Finally, we illustrate what it means to detect violations at the earliest possible time: for a given set of rule and event stream, a violation is detected at the earliest possible time t if and only if at all times before t, the stream can be extended by future events to satisfy the set and all times at and after t, the stream cannot be extended by future events to satisfy the set. To detect violations at the earliest possible time is the *early violation detection* problem.

Given this framework and problem, a fundamental question to address is whether or not it is possible to automate early violation detection. We show that this is impossible by showing the related problem of finite satisfiability is computationally undecidable (Chapter 6). To prove this, we use a modified rule language called Datalog$^+$ and reduce the empty-tape Turing machine halting problem (known to be undecidable) to determining finite satisfiability for a set of Datalog$^+$ rules constructed from a Turing machine. Then, we show finite satisfiability for a set of Datalog$^+$ rules can be reduced to finite satisfiability for a set of rules in our language. This result indicates that early violation detection, though desirable, is fundamentally intractable, and thus the problem requires simplfying assumptions to be solved algorithmically.

To establish feasible cases of early violation detection, we study subproblems where the sets of rules or event streams have certain restrictions, including when the rules constrain exclusively event timestamps (and not event data), when only a single rule is considered, whether or not the set of rules is acyclic, i.e., whether or not the dependencies between rules in the target set form a graph with a cycle, and how many events per second are in the stream. For each of these subproblems, we develop or improve techniques for early violation detection.

Given our focus on quantitative time constraints, a natural subclass of rules we consider are those that only constrain event timestamps and not event data, i.e., dataless rules; we describe how early violation detection is possible through translation from dataless rules to LTL formulas to finite state machines and provide two translations that improve on existing techniques (Chapter 3). For the first translation, we use a graph representation to characterize "acyclic" and "singly-linked" rules; the tree structure of acyclic, singly-linked, dataless rules, allows us to translate the time constraints (if any) on each pair of events in the rule separately, then combine these into a single translation. Building on this translation, we devise a second translation for arbitrary, singly-linked, dataless rules by leveraging a decomposition of quantitative time constraints into all their possible gaps and orderings, which reduces the problem to the acyclic case. The size of the output of these two translations is an LTL formula whose size is at most single- or double-exponential, respectively, in the size of the input rule, both improving on the hyper-exponential size of the best known translation.

Using constraints on event data, e.g., requiring that the same user who makes a request also receives approval, allows for more expressive rules, but also makes early violation detection more difficult because the values of data variables are not known in advance, and thus finite state machines alone cannot be used to detect violations; to address this, we also study rules with data variables, developing algorithms based on assignments to these variable (Chapter 4). We define data structures to track potential violations and present algorithms to update these data incrementally as new events arrive from the stream. Notably, we use a chase process, a well-known technique in database systems, to reason about dependencies between rules and potential violations, and a satisfiability test on linear inequalities to compute "deadlines", the earliest time at which a violation is inevitable, that mark permanent violations. To ensure the chase terminates, we restrict these algorithms to acyclic sets of rules, i.e., sets of rules whose dependencies form a directed acyclic graph, a common restriction when applying the chase process. To evaluate the effectiveness of these algorithms, we implemented them in a software runtime monitor and tested them on a variety of rule sets and event streams to measure their feasibility, benefits, and limitations. The evaluation shows that early violation detection can be beneficial by greatly reducing the number of events that must be processed to identify violating streams, and show that our techniques are feasible for medium-scale systems, e.g., thousands of events per second.

Fortunately, our techniques for acyclic sets of rules with data variables can be extended to include aggregation functions, a common features of business rules that aggregate properties of groups of events, without affecting the decidability of early violation detection. In Chapter 5, we develop a framework for classifying and specifying time windows on streams and for defining aggregation functions over these windows in our

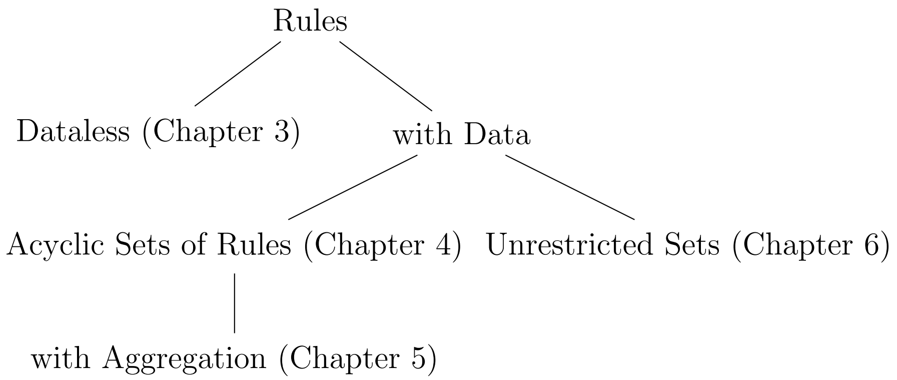
Rules
Dataless (Chapter 3)
with Data
Acyclic Sets of Rules (Chapter 4)
Unrestricted Sets (Chapter 6)
with Aggregation (Chapter 5)

Figure 1.1: Dissertation organization.

rule language. We show that these features, along with the restriction that the target stream has at most one event per timestamp, still allow early violation detection for acyclic sets of rules. To integrate aggregation into our existing algorithms, we provide two approaches. First, we present an encoding of aggregation rules as Datalog$_{\mathbb{Z}}$ programs, providing a means of evaluating rules with aggregation within a Datalog framework. Alternatively, we develop techniques to rewrite aggregation expressions over time windows as Presburger arithmetic formulas, for which decision procedures exist. This rewriting allows us to apply a similar chase process and satisfiability testing to acyclic sets of rules with aggregation functions, enabling early violation detection.

We organize the dissertation according to the various subclasses of rules considered. First, in Chapter 2, we present definitions of core concepts. Then, the chapters are organized by the the subclasses of rules or sets of rules they consider, as shown in Fig. 1.1. Chapter 3 studies early violation detection for dataless rules, Chapter 4 studies rules with data variables, Chapter 5 adds aggregation functions over time windows, and Chapter 6 considers unrestricted sets of rules.

Chapter 2

Preliminaries

In this chapter, we define the concepts used throughout the dissertation. First, in Section 2.1, we introduce an event stream model for sequences of events, ordered by timestamps and carrying data, called “enactments”. In Section 2.2 we introduce a language for specifying constraints on enactments, called “business process rules” or “rules” for short. Rules use Datalog syntax [19], with the addition of (i) syntax for separating an event’s data from its timestamp, (ii) inequality predicates and arithmetic on event timestamps, and (iii) existential variables in the rule head, and (iv) multiple atoms in the rule head. Our interpretation of rules differs from Datalog in that we treat rules as constraints, not as programs that generate events. We define the semantics of rule satisfaction and violation, then define what it means to detect violations at the earliest possible time. In Section 2.3, we illustrate the basic concepts of our approach to violation detection with an example.

2.1 Events and Workflow Enactments

To describe events, we assume the following pairwise-disjoint, infinite sets:

- $\mathcal{E}$ of event names, typically single words in typewriter font with a capitalized first letter, e.g., `Request`, `Payment`, ...

- $\mathcal{A}$ of attribute names, typically single, italicized, lowercase words, e.g., *user*, ...
- **I** of *enactment identifiers*, or simply ID's, to identify the enactment to which an event belongs,
- A data domain $\mathcal{D}$ of uninterpreted constants, with the equality ($=$) and non-equality predicate ($\neq$), and
- Discrete timestamps. Without loss of generality, we use the natural numbers $\mathbb{N}$ as timestamps, with the standard addition operation ($+$) and equality, non-equality, and inequalities predicates ($=$, $\neq$, $<$, $\ldots$) as well as the integers $\mathbb{Z}$ for related technical development.

We study early violation detection in the context of monitoring event streams generated by enactments of workflows. In a workflow, activities are atomic units of work, e.g., a `Request` arrives and is processed. We treat the completion of an activity as an event with no duration.

Definition. An *event type* $E(a_1, \ldots, a_n)$, where $n \geqslant 0$, has an event name E from $\mathcal{E}$ and a fixed set of *attributes* a_1, $\ldots$, a_n, each from $\mathcal{A}$. An event type is *dataless* if it has no attributes, i.e., $n = 0$. An *event (instance)* of an event type $E(a_1, ..., a_n)$ is a named tuple $E(c_1, ..., c_n)@t$ where c_i is a value from $\mathcal{D}$ for each attribute a_i and t is a timestamp from $\mathbb{N}$.

For a given workflow, its schema is a set of event types for its component activities, each with the activity's name and data attributes. We assume that the schema is fixed and known to the monitor *a priori*. Events are generated by the completion of the activities in a workflow, so each event carries with an enactment identifier from **I**.

Definition. An *enactment* of a workflow is a finite set η of events, such that (i) each event has the same enactment ID, (ii) η has exactly one special event of type *START* that marks its beginning and at most one *END* event that marks its completion, (iii) the timestamp of the *START* event is less than that of all other events in η, and (iv) the timestamp of the *END* event, if it occurs, is greater than that of all other events in η.

Workflow enactments are updated by the completion of new events, grouped by timestamp: a *batch* for an enactment η is a finite set Δ of events such that (i) all events in Δ have the same timestamp, denoted as ts_Δ, greater than the timestamps of all events in η, (ii) for each event e in Δ, the ID of e is the ID of η, (iii) Δ has a *START* event or η has a *START* event, but not both, and (vi) if an *END* event is in η, no events are in Δ.

2.2 Rules

We express constraints in a language first introduced in [20], named here *(business process) rules*, or simply *rules.* The rule language It uses a Datalog-like syntax, but unlike Datalog, the rule syntax separates an event's data from its timestamp, allows multiple atoms and existential variables in the rule head, and gap atoms with inequalities on time variables. First, we provide the rule syntax, then we define what it means for an enactment to satisfy or violate a rule or set of rules.

We start with atomic formulas. We use the existing sets from the previous section and define the following infinite set, disjoint from the others: $\mathcal{V}$ of *variables*, typically lowercase letters, e.g., $a, b, c, x, y, z, \ldots$. We use the notation $\bar{x}$ to denote vectors of variables. An *event atom* is an expression "$A(v_1, ..., v_n)@x$" where $A(\mathrm{c}_1, ..., \mathrm{c}_n)$ is an event type, $v_1, ..., v_n, x$ are variables in $\mathcal{V}$ or constants in $\mathcal{D}$ or $\mathbb{N}$. If x is a variable, x is a *timestamp variable.* Additionally, constant addition is allowed on timestamps, i.e., $v + c$ is a valid term for a timestamp, with variable v and constant $c \in \mathbb{Z}$. When an event type E is dataless, the event atom is written as $E@x$ instead of $E()@x$. A *gap atom* (inspired by [21]) is an expression "$x{\pm}\epsilon\,\theta\, y$" where x, y are timestamp variables, ϵ (the gap) is a constant in $\mathbb{Z}$, and $\theta \in \{<, \leqslant, \geqslant, >, =\}$ is an equality or inequality predicate. The set of variables in a set of atoms φ is denoted $var(\varphi)$. An event atom $E(\dots)@x$ expresses that an instance of E happens at time x, e.g., $\texttt{Request}(u)@x$ denotes a `Request` for some user u at time x.

A gap atom $x \leqslant_n y$ means $x + n \leqslant y$, i.e., $x + n$ no later than time y. Similarly, $x \geqslant_n y$ means $x + n \geqslant y$, i.e., $x + n$ is no earlier than time y. Some gap atoms do not impose an ordering: when $n > 0$, $x \geqslant_n y$ indicates y is at most n time units after x and potentially simultaneous with or before x. Note that for each positive $n \in \mathbb{N}$, $x \geqslant_n y$ does not imply $x \geqslant y$.

Example 2.1 : Consider the set of atoms "$\texttt{Request}(u)@x$, $\texttt{Schedule}(u)@y$, $x \leqslant_4 y, x \geqslant_6 y$". Intuitively, it selects a `Request` event and a `Schedule` event from the same enactment by the same user u that follows the `Request` by four, five, or six timestamps. ▮

Definition. A *rule* is an expression "$\varphi \rightarrow \psi$" where the *body* φ and the *head* ψ are finite, possibly empty, sets of event and gap atoms such that the body is *closed*: each variable in φ occurs in some event atom in φ, and the head is closed with respect to the head and body: each variable in ψ occurs in some event atom in $\varphi \cup \psi$. Furthermore, if the body of a rule $\varphi \rightarrow \psi$ has no atoms, the rule is written *true* $\rightarrow \psi$, with the natural semantics.

Example 2.2 : The following rule r_0 requires for each pair of `Request` and subsequent `Schedule` events from the same user u, there is a subsequent `Payment` no later than three days after the `Schedule` by that user:

$$r_0 : \texttt{Request}(u)@x, \texttt{Schedule}@y, x \leqslant_0 y \rightarrow \texttt{Payment}(u)@z, y \leqslant_0 z, y \geqslant_3 z$$ ▮

Rule satisfaction is defined with respect to "assignments" from values in the event stream, which come from $\mathcal{D}$ and $\mathbb{N}$, to rule variables. An *assignment* is a mapping from rule variables to values in $\mathcal{D} \cup \mathbb{N}$. An assignment is *complete* if it is a total mapping for the variables in a given set of atoms. An assignment β *extends* an assignment α if $\alpha \subseteq \beta$.

An enactment η *satisfies* an event atom $A(v_1, ..., v_n)@x$ for the event $A(\mathrm{c}_1, ..., \mathrm{c}_n)$ with an assignment μ if (1) μ is defined for $v_1, ..., v_n, x$, and (2) $A(\mu(v_1), \ldots, \mu(v_n))@\mu(x)$ is an event in η. *Satisfaction* for gap atoms is defined naturally. A set of atoms is called a *constraint*: two constraints ϕ and ϕ' are *equivalent* if for each enactment η and assignment σ, $\eta \models \phi[\sigma]$ iff $\eta \models \phi'[\sigma]$. An enactment η *satisfies* a set of atoms ϕ with an assignment μ if η satisfies every atom in ϕ with μ.

We can now define the central notions of how an enactment satisfies or violates a rule or a set of rules. Informally, this is done by matching assignments for the body variables with those for the head variables. Let $r : \varphi \to \psi$ be a rule and η an enactment. Then, η *satisfies* $\varphi \to \psi$ if for every assignment μ such that η satisfies φ with μ, there is an assignment β that extends μ such that η satisfies ψ with β. Alternatively, η has a *potential violation* of a rule $\varphi \to \psi$ with witness μ if η satisfies φ with μ and there is no assignment β that extends μ such that η satisfies ψ with β.

A potential violation becomes inevitable, or permanent, when it cannot be resolved in any possible future of the current enactment. The enactment η has a *(permanent), violation* of a rule r with a witness μ if it has a potential violation with μ and for every sequence $\Delta_1, ..., \Delta_n$ of batches of (future) events (Δ_i is a batch for $\eta \cup (\cup_{j<i} \Delta_j)$ for each $1 \leqslant i \leqslant n$), there is no assignment β that extends μ such that $\eta \cup (\cup_{i=1}^{n} \Delta_i)$ satisfies ψ with β. Finally, the enactment η has a *(permanent) violation* of a rule r if it has a (permanent) violation of r with some assignment μ.

Example 2.3 : Consider the rule r_0 in Example 2.2 and an enactment with exactly two events: `Request`(*Alice*)@10 and `Schedule`(*Alice*)@12. The assignment $\alpha : \{u \mapsto \mathit{Alice}, x \mapsto 10, y \mapsto 12\}$ satisfies r_0's body and indicates a potential violation of r_0 at times 12, 13, and 14, because a `Payment` event is required by the rule head but has not happened. Assuming no more events happen, the assignment is the witness of a permanent violation at time 16 for the enactment, because no `Payment` event can happen after time 15 to match α. ∎

We further extend the notion of violation to a set of rules; notably such violations may not have a witness. An enactment η has a *potential violation* of a set of rules $\mathcal{R}$ if there is some rule $r \in \mathcal{R}$ that has a potential violation for η. An enactment η has a *(permanent) violation* of a set of rules $\mathcal{R}$ if there is a potential violation of $\mathcal{R}$ in η and for every sequence $\Delta_1, ..., \Delta_n$ of batches of future events, $\eta \cup (\cup_{i=1}^{n} \Delta_i)$ has a potential violation of some $r \in \mathcal{R}$.

2.3 An Opportunity for Early Violation Detection

We illustrate the problem of detecting violations of a single rule and motivate an approach that calculates the earliest time a violation is inevitable, called a “deadline”. We use an example workflow enactment from an Infrastructure-as-a-Service (IaaS) provider that offers commodity machines for cloud computing rental. The service is managed by a workflow with the following activities (in the typewriter font): the user makes a `Request` for a machine through an account and the provider grants `Approval` to the user. Then, the user can `Reserve` a machine for their account, make a `Payment` with their account and `Launch` the machine. The completion of each activity generates an event; events for the same rental service instance form an enactment. Event of the same type have the same attributes, and thus can be organized in a relational database. Fig. 4.1 shows a database S_9 at time 9, with eight events from two enactments with ids π_1 and π_2. For example, the first row of the `Request` table shows a `Request` event with enactment id π_1 from user Alice with account $a3$ at time 1.

The IaaS provider monitors its enactment against specified rules; these may measure service availability, quality, etc.; for example, a requirement r_1 states “when a user’s `Request` is approved within 7 days and the machine is `Reserve`d within 7 days of `Approval` by the same account as the request, the user should make a `Payment` for the

Request

ID	*user*	*account*	ts
π_1	Alice	a3	1
π_1	Alice	a4	3
π_2	Bob	b6	7

Approval

ID	*user*	ts
π_1	Alice	6

Reserve

ID	*user*	*account*	ts
π_1	Alice	a4	8
π_1	Alice	a3	9

Payment

ID	*user*	*account*	ts
π_1	Alice	a3	8
π_1	Alice	a4	9

Launch

ID	*user*	*account*	ts

Figure 2.1: Database S_9 with events from two enactments π_1 and π_2.

machine through that account within 3 days of `Approval` and `Launch` it within 7 days of `Reserve` and 4 days of `Payment`." Note that events are generated by both the provider and the user. We write this requirement as a rule:

$$r_1 : \ \texttt{Request}(u, a)@x, \texttt{Approval}(u)@y, x \leqslant y \leqslant x+7, \texttt{Reserve}(u, a)@z, y \leqslant z \leqslant \ y+7$$

$$\rightarrow \texttt{Payment}(u, a)@w, \texttt{Launch}(u, a)@v, y \leqslant w \leqslant y+3, z \leqslant v \leqslant z+7, v \leqslant w + 4$$

The core requirement for detecting a violation of this rule is checking whether each assignment for the body variables u, a, x, y, z satisfying φ has a matching assignment for the head variables u, a, w, v satisfying ψ. In order to detect violations incrementally, we store assignments that satisfy subsets of the rule body or head. For example, assignment μ_{10} is generated by the body events $\texttt{Request}(\pi_1, [\text{Alice}, a4])@3$, $\texttt{Approval}(\pi_1, [\text{Alice}])@6$, and $\texttt{Reserve}(\pi_1, [\text{Alice}, a4])@8$. Then, $\mu_{10} : \{\text{ID} \mapsto \pi_1,\ u \mapsto \text{Alice},\ a \mapsto a4,\ x \mapsto 3,\ y \mapsto 6,\ z \mapsto 8\}$ makes φ true.

ψ has six variables u, a, y, z, w, v, but the `Payment` and `Launch` events only supply values for the four "event variables" u, a, w, v. We consider assignments for ψ in the same manner as for φ but ignoring y and z. The `Payment` events at times 8 and 9 (Fig. 4.1) create assignments β_1: $[\pi_1, \text{Alice}, a3, 8, -]$ and β_2: $[\pi_1, \text{Alice}, a4, 9, -]$.

We aim to detect violations at the earliest possible time. In Fig. 2.2, three events create a potential violation μ_{10}. The violation is certain when the end of the enactment π_1 arrives, after which no more events π_1 can happen, and thus there will be no more assignments to match μ_{10}. However, given the rule's constraints $y \leqslant w \leqslant y+3$ and $z \leqslant v \leqslant z+7$, and $\mu_{10}(y)=6$ and $\mu_{10}(z)=8$, to satisfy the rule w.r.t. μ_{10}, there must be an assignment extending μ that satisfies ψ; i.e., two events $\texttt{Payment}(\pi_1, [\text{Alice, a4}], t_1)$ and $\texttt{Launch}(\pi_1, [\text{Alice, a4}], t_2)$ with timestamp constraints $6 \leqslant t_1 \leqslant 6+3=9$, $8 \leqslant t_2 \leqslant 8+7=15$, and $t_2 \leqslant t_1+4$ must happen.

Then, the violation is inevitable at time 8 if no `Payment` event arrives with a timestamp for w to extend μ_{10}. Furthermore, 8 is also the earliest time this violation is permanent; before then, a `Payment` event can arrive.

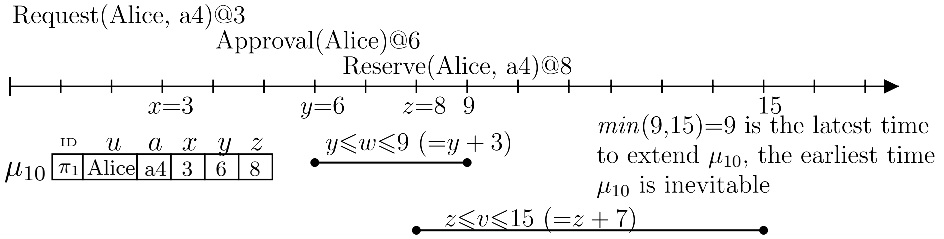
Request(Alice, a4)@3
Approval(Alice)@6
Reserve(Alice, a4)@8
x=3
y=6
z=8
9
15
ID
u
a
x
y
z
μ_{10}
π_1
Alice
a4
3
6
8
$y \leqslant w \leqslant 9$ (=$y+3$)
min(9,15)=9 is the latest time to extend μ_{10}, the earliest time μ_{10} is inevitable
$z \leqslant v \leqslant 15$ (=$z+7$)

Figure 2.2: Deadline for extending potential violation, body assignment μ_{10}

Fig. 2.3 shows the result of a `Payment` event at time 9. This creates the assignment β_2, leading to a new deadline calculation for the potential extension of μ_{10} by β_2.

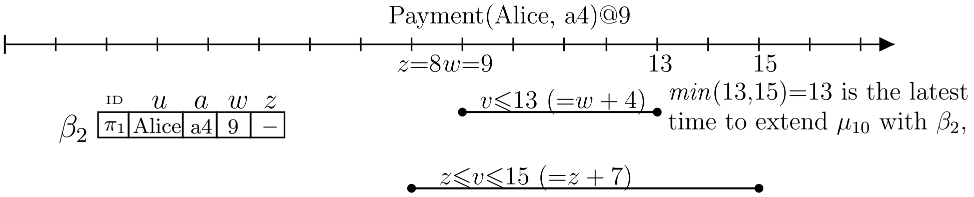
Payment(Alice, a4)@9
z=8w=9
13
15
ID
u
a
w
z
β_2
π_1
Alice
a4
9
–
$v \leqslant 13$ (=$w+4$)
min(13,15)=13 is the latest time to extend μ_{10} with β_2,
$z \leqslant v \leqslant 15$ (=$z+7$)

Figure 2.3: Deadline for extending β_2 as match for μ_{10}

With the above technical definitions and illustration of the problem, we can now state the main focus of this dissertation: *Given a rule or a set of rules, monitor a workflow enactment for violations of the rule or set of rules, respectively, as the enactment is updated with new events.*

Chapter 3

Rules without Data

In this chapter, we develop techniques for automatically generating monitors to detect violations of rule without data. This approach involves two steps, translating: (1) dataless rules to formulas in linear temporal logic (LTL) on finite traces, and (2) LTL formulas to finite state machines. Since algorithms exist for step (2) [22–24], we focus on step (1), i.e., mapping rules to equivalent LTL formulas. Then, a finite state machine can process an enactment incrementally and report whether or not the enactment is a prefix in the machine's language. Kamp's Theorem [25] implies that each dataless rule has an equivalent LTL formula. A recent proof of Kamp's Theorem [26] provides key pieces of a translation algorithm, though it produces formulas hyper-exponential in the size of the input rule. We present and establish the correctness of two translation techniques for "singly-linked", dataless rules. In the first, we use a graph representation for a recursive translation of "acyclic", singly-linked, dataless rules. In the second translation, we use ordering constraints on timestamp variables resembling those in [26] to translate singly-linked, dataless rules. The second translation covers a larger class of rules, but produces larger LTL formulas. Finally, we provide bounds for the size of formulas produced by our techniques and show that they are smaller than those produced by the translation in [26].

The organization of this chapter is as follows: Section 3.1 motivates our approach to violation detection for dataless rules with an example. Section 3.2 presents the LTL language used for translation and the notion of equivalence between LTL formulas and dataless rules. Section 3.3 presents our translation techniques first for gap atoms, then for singly-linked, acyclic rules, and proves their correctness. Section 3.4 extends them to arbitrary singly-linked rules. Finally, Section 3.5 compares the size of formulas produced by our translation with a translation derived from Kamp's Theorem and covers related work.

3.1 Monitoring with Finite State Machines

First we motivate our approach by showing how a finite state machine can monitor an enactment for a given rule.

Example 3.1 : Consider an Infrastructure-as-a-Service (IaaS) provider that rents commodity servers to clients as a service. The service is performed by processes (in the typewriter font), some initiated by clients, others by the service provider, including the following: a client may `Request` access by providing a description of their desired machine(s) and the provider uses the `Schedule` process to reserve a specific machine. The completion of a process generates an event of the same name, along with a timestamp. The service includes a business rule PromptSchedule to ensure service quality:

> *If a `Request` arrives, a `Schedule` instance must be completed on the same day or in the next three days.*

This rule can be specified as:

$$\texttt{Request}@x \;\rightarrow\; \texttt{Schedule}@y,\; x \leqslant y,\; y \leqslant x + 3$$

where $@x$ and $@y$ indicate time instants `Request` and `Schedule` were performed, respec-

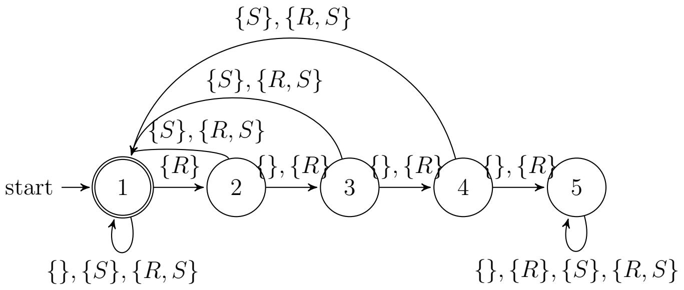

Figure 3.1: A finite state machine for the PromptSchedule rule

tively.

The finite state machine in Fig. 3.1 monitors this rule. The machine transitions once per day, changing state based on the events observed in the enactment in the previous day. The machine's alphabet is the powerset of the rule's event types: `Request` (R) and `Schedule` (S). The start state (node 1) is accepting because initially no `Request` requiring a subsequent `Schedule` has been observed. The machine moves from state 1 to state 2 when `Request` takes place without a simultaneous `Schedule`. From state 2, for each subsequent day that `Schedule` doesn't occur, the machine progresses through states 3 and 4, implicitly counting the number of days since the unmatched `Request`. Although the rule has not been permanently violated in states 2, 3, or 4, these states are not accepting because no matching `Schedule` event has been observed; the distinction between an enactment permanently or temporarily violating a rule is necessary for reasoning about when to report violations. If a matching `Schedule` event occurs from states 3 and 4, all recent (within 3 days) `Request` events are matched and the machine moves to state 1. If the machine is in state 4 and the following day fails to contain a `Schedule`, the machine enters the sink state 5, as the enactment has violated PromptSchedule. ▮

In the subsequent technical development of this chapter, we translate dataless rules to LTL with past operators for finite traces. LTL can be converted into finite state machines using existing algorithms [22, 23], which can be used for violation detection. Accordingly, violation detection requires simply transitioning a finite state machine—

a much simplified task. To proactively avoid violations, it is sufficient to check the reachability of accepting states from the current state. To monitor a given set of rules, one constructs a product machine using the machine from each rule. Summarizing the above discussions, Our central technical problem is the following: *Given a dataless rule for an enactment, construct an equivalent LTL formula.*

3.2 Dataless Enactments and Linear Temporal Logic

The section describes how the enactments can be mapped to LTL traces. First, we introduce the model of events without data. Then, we review the syntax and semantics of a past- and future-time linear temporal logic on finite traces. Finally, we discuss the notion of equivalence between dataless rules and LTL formulas needed to state the correctness of a translation.

We focus on time gaps between events, ignoring other data: each event is represented only by its name and a timestamp. Thus, an enactment can be faithfully represented by a set of unary relations over the domain of timestamps.

Example 3.2 : Timestamps in our model are given as the number of days into the year 2023 in Fig. 3.2. For example, the date 2023-01-04 receives the timestamp 4, 2023-02-04 the timestamp 35. The initial `Request` events in the dataless enactment have timestamps 1, 6, 25, and 30, and the initial `Schedule` events have timestamps 3, 4, 8 and 42, etc. Fig. 3.3 is the initial segment of the trace of the enactment in Fig. 3.2. ▮

Example 3.3 : Recall that for dataless events, e.g., `Request`, we write event atoms without parentheses: `Request` instead of `Request()`. The set of atoms ϕ: "`Request`@x, `Schedule`@$y, x \leqslant_0 y, x \geqslant_3 y$" selects two events: a `Request` and a matching `Schedule` between 0 and 3 days after the `Request`. Consider the enactment η in Fig. 3.2. For this enactment, we have two assignments for the variables in ϕ: $\sigma_1 : \{x \mapsto 1, y \mapsto 3\}$

`Request`

iid	*#machines*	*timestamp*	...
000	5	2023-01-01	
001	5	2023-01-06	
002	10	2023-01-25	
003	5	2023-01-30	
...			

`Schedule`

iid	*machine*	*timestamp*	...
100	445	2023-01-03	
101	446	2023-01-04	
102	447	2023-01-06	
103	447	2023-02-11	
...			

`Compute`

iid	*machine*	*timestamp*	...
400	445	2023-01-05	
401	445	2023-01-18	
...			

`Payment`

iid	*amount*	*timestamp*	...
500	$150	2023-01-15	
501	$155	2023-01-30	
...			

Figure 3.2: Events with data generated by four activities.

i	0	1	2	3	4	5	6	...
$\pi_\eta[i]$		`Request`		`Schedule`	`Schedule`	`Compute`	`Request`, `Schedule`	...

Figure 3.3: Trace π_η of the enactment in Fig. 3.2

.

and $\sigma_2 : \{x \mapsto 1, y \mapsto 6\}$. Given the rule semantics, $\eta \models \phi[\sigma_1]$ because $1 \in \eta(\texttt{Request})$, $3 \in \eta(\texttt{Schedule})$, $1 \leqslant_0 3$, and $1 \geqslant_3 3$. However, $\eta \not\models \phi[\sigma_2]$ because $1 + 3 \not\geqslant 6$ and $6 \notin \eta(\texttt{Schedule})$. ▮

Example 3.4 : The following rule r_0 requires each pair of `Request` and subsequent `Schedule` events to be followed within 3 days by `Payment`:

$$r_0 : \ \texttt{Request@}x, \texttt{Schedule@}y, x \leqslant_0 y, \rightarrow \texttt{Payment@}z, y \leqslant_0 z, y \geqslant_3 z$$

The enactment η in Example 3.2 does not satisfy r_0: for the `Request` instance at time 1 (*iid* 000) and the `Schedule` instance at time 3 (*iid* 100), corresponding to the rule body assignment $\sigma_1(x \mapsto 1, y \mapsto 3)$, there is no `Payment` instance with timestamp between 3 and 6, i.e., no matching assignment for the rule head. ▮

Given a set of rules R, the discussions in Section 3.1 suggest determining if an enactment satisfies R using finite state machines derived from LTL for finite traces. We define below the LTL operators used in our translation. We treat each activity p in S as a propositional variable. LTL formulas for S are defined recursively as follows:

$$\phi := p \mid \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{X}\phi \mid \mathsf{X}^{-1}\phi \mid \mathsf{F}\phi \mid \mathsf{P}\phi$$

where $p \in S$, *true* and *false* are Boolean constants, and $\neg$ (not) and $\wedge$ (and) are Boolean operators. The standard Boolean abbreviations $\vee$ (or) and $\rightarrow$ (implies) are used as well. The temporal operators X (*next*) and F (*future*) are common in future-time LTL [7], while P (*past*) and X^{-1} (*yesterday*) (sometimes written as Y [27]) are common in past-time LTL [28]. The following notion is used for convenience: X^k ($k \in \mathbb{Z}$) means k consecutive X operators when $k > 0$, k consecutive X^{-1} operators when $k < 0$, and 0 instances of the X operator when $k = 0$.

LTL formulas are satisfied by "traces" defined as follows. An *interpretation* is a mapping from a set of propositions S to $\{\mathit{true}, \mathit{false}\}$. A *trace* π is a (finite) sequence of interpretations, with length $\mathit{len}(\pi)$, and for $0 \leqslant i \leqslant \mathrm{len}(\pi) - 1$, i is a *(time) instant* in π and $\pi[i]$ denotes the i^{th} interpretation in π. For an instant i such that $0 \leqslant i \leqslant \mathit{len}(\pi) - 1$, and an LTL formula ϕ for S, we say π *satisfies* ϕ *at* i, denoted $\pi, i \models \phi$, if one of the following is true (the cases for Boolean constants and operators are standard and thus omitted):

- $\pi, i \models p$ if $\pi[i](p) = \mathit{true}$,
- $\pi, i \models \mathsf{X}\phi$ if $i < \mathit{len}(\pi) - 1$ and $\pi, i+1 \models \phi$,

- $\pi, i \models \mathsf{X}^{-1}\phi$ if $i > 0$ and $\pi, i{-}1 \models \phi$,
- $\pi, i \models \mathsf{F}\phi$ if $\exists j$, $i \leqslant j \leqslant len(\pi){-}1$ and $\pi, j \models \phi$, and
- $\pi, i \models \mathsf{P}\phi$ if $\exists j$, $0 \leqslant j \leqslant i$ and $\pi, j \models \phi$

The standard LTL abbreviation G (global) is used as well. Intuitively, the semantics indicate that π satisfies $\mathsf{X}^k\phi$ or $\mathsf{X}^{-\mathsf{k}}\phi$ at an instant i if ϕ is satisfied at the k^{th} following (resp., preceding) instant from i, and π satisfies $\mathsf{F}\phi$ (or $\mathsf{P}\phi$) at an instant i if ϕ is satisfied at i or an upcoming (resp., previous) instant.

Enactments are closely related to finite traces. Given an enactment η, let κ be the largest timestamp in η and 0 if η is empty. The following mapping converts enactments to and from traces: let η be an enactment with event type S. The *trace* π_η is the sequence $\pi_\eta[0]\cdots\pi_\eta[\kappa]$ where for each $i \in [0..\kappa]$ and each $p \in S$, $\pi_\eta[i](p) = \mathit{true}$ if $i \in \eta(p)$, and *false* otherwise. Conversely, for each trace π, the enactment η_π is defined as follows: for each $i \in [0...len(\pi) - 1]$ and each $p \in S$, $i \in \eta_\pi(p)$ if $\pi[i](p) = \mathit{true}$.

Based on the above enactment-trace mapping, we conveniently use the notation $\eta, i \models \phi$ to mean $\pi_\eta, i \models \phi$ and can state the main technical problem:

Given a set R of rules for event types S, is there an LTL formula ϕ over S such that for each enactment η of S, $\eta \models R$ iff $\eta, 0 \models \phi$?

3.3 Recursive Translation of Acyclic Constraints

In this section, we develop techniques to translate dataless subclasses of constraints and rules into equivalent LTL formulas and establish the main technical result of the chapter (Theorem 3.5). First, we describe a mechanism for mapping gap atoms over two variables to LTL operators. Then we use a graph representation of constraints to define and translate connected, acyclic constraints. We state a key lemma (Lemma 3.2) concerning the correctness of this translation, which associates a satisfying assignment for a constraint with an instant in the trace when the constraint's translation is satisfied. We then translate arbitrary acyclic constraints and establish its correctness as another lemma (Lemma 3.4). Finally, we provide a translation function for the subclass of singly-linked, acyclic, dataless rules using the previous results, and state its correctness (Theorem 3.5).

Example 3.5 : Consider the constraint RENTAL that captures the typical behavior of the IaaS service:

$$\begin{array}{l}\texttt{Request}@x,\ \texttt{Schedule}@y,\ \texttt{Compute}@z,\ \texttt{Terminate}@w,\\ x \leqslant_1 y,\ \ x \geqslant_{10} y,\ \ y \geqslant_5 z,\ \ z \leqslant_0 w\end{array}$$

Intuitively, RENTAL selects timestamps x, y, z, and w for `Request`, `Schedule`, `Compute`, and `Terminate` events, resp., that satisfy the following three conditions:

(i) `Schedule` occurs at least one day but no more than 10 days after `Request`,

(ii) `Compute` occurs no later than 5 days after `Schedule`, and

(iii) `Compute` occurs before or simultaneously with `Terminate`.

These conditions are observed in a trace as follows: Condition (i) is observed in a trace if the proposition `Request` holds at an instant x and the proposition `Schedule` holds at instants $x{+}1, x{+}2, \ldots,$ or $x{+}10$. We rewrite the latter using next LTL operators (X) as: $\mathsf{X}\,\texttt{Schedule}$, $\mathsf{X}^2\,\texttt{Schedule}$, $\ldots$, or $\mathsf{X}^{10}\texttt{Schedule}$ to hold at x. Using Boolean operators produces a translation of condition (i): "$\texttt{Request} \wedge \bigvee_{1 \leqslant j \leqslant 10} \mathsf{X}^j \texttt{Schedule}$".

Condition (ii) is observed if `Schedule` holds at some instant y and `Compute` holds at or before $y{+}5$. We rewrite the second statement with next and past LTL operators: $\mathsf{X}^5\mathsf{P}\,\texttt{Compute}$ holds at y. Thus, a translation of condition (ii) is: "$\texttt{Schedule} \wedge \mathsf{X}^5\mathsf{P}\,\texttt{Compute}$".

Condition (iii) is observed if `Compute` holds at z and `Terminate` holds at z or a future instant. Rewriting the latter with the future LTL operator, condition (iii) is present when `Compute` and $\mathsf{F}\,\texttt{Terminate}$ hold at z. Thus, a translation of condition (iii) is "$\texttt{Compute} \wedge \mathsf{F}\,\texttt{Terminate}$". ▮

To generalize from these examples we define the following function for LTL formula construction from gap atoms. We use $\equiv$ to denote equivalence between two sets of gap atoms; we say $\phi_1 \equiv \phi_2$ for sets ϕ_1 and ϕ_2 if $\mathit{var}(\phi_1) = \mathit{var}(\phi_2)$ and for each assignment σ to $\mathit{var}(\phi_1)$, $\phi_1[\sigma]$ and $\phi_2[\sigma]$ are both true or both false.

Definition : For a set of gap atoms ϕ over two variables x and y, let $\mathrm{Gap}_{x,y}(\phi)$ be the following operator combinations:

$$\mathrm{Gap}_{x,y}(\phi) = \begin{cases} \mathsf{X}^n\mathsf{F} & \text{if } \phi \equiv \{x \leqslant_n y\} \text{ for some } n \in \mathbb{Z} \\ \mathsf{X}^m\mathsf{P} & \text{if } \phi \equiv \{x \geqslant_m y\} \text{ for some } m \in \mathbb{Z} \\ \bigvee_{n \leqslant j \leqslant m} \mathsf{X}^j & \text{if } \phi \equiv \{x \leqslant_n y, x \geqslant_m y\} \text{ for some } n, m \in \mathbb{Z} \end{cases}$$

The Gap function translates the conditions in Example 3.5 as follows: for condition (i), the proposition $\texttt{Request}$ is used for the atom $\texttt{Request}@x$. Applying $\mathrm{Gap}_{x,y}$ to $\{x \leqslant_1 y, x \geqslant_{10} y\}$ produces the operators $\bigvee_{1 \leqslant j \leqslant 10} \mathsf{X}^j$, and the proposition corresponding to variable y, $\texttt{Schedule}$, is placed after these operators: $\texttt{Request} \wedge \bigvee_{1 \leqslant j \leqslant 10} \mathsf{X}^j\ \texttt{Schedule}$.

The following lemma shows that each constraint with two variables is equivalent to some constraint with at most two gap atoms, i.e., the input required for the Gap function.

Lemma 3.1 : Let ϕ be a set of gap atoms over two variables. Without loss of generality, let $\mathit{var}(\phi) = \{x, y\}$. Then ϕ is equivalent to a set in one of the following three forms:

- $\{x \leqslant_n y\}$, for some $n \in \mathbb{Z}$,
- $\{x \geqslant_m y\}$, for some $m \in \mathbb{Z}$, or
- $\{x \leqslant_n y, x \geqslant_m y\}$, for some $n, m \in \mathbb{Z}$.

Proof: First, note that a gap atom with exactly one variable is either a tautology or a contradiction, thus equivalent to either $\{x \leqslant_0 x\}$ or $\{x \leqslant_1 x\}$. Second, observe that for all $m \in \mathbb{Z}$, $y \leqslant_m x$ iff $x \geqslant_{-m} y$, and $y \geqslant_m x$ iff $x \leqslant_{-m} y$. These equivalences allow all gap atoms in ϕ to be written with x on the left of the predicate and y on the right.

Next, note that for all $n, m \in \mathbb{Z}$ with $n \leqslant m$, $x \leqslant_m y$ implies $x \leqslant_n y$, and $x \geqslant_n y$ implies $x \geqslant_m y$. If ϕ contains $x \leqslant_n y$ and $x \leqslant_m y$ where $n \leqslant m$, then ϕ is equivalent to $\phi - \{x \leqslant_n y\}$. A similar statement holds for $\geqslant$-atoms. Repeating these eliminations of atoms in ϕ yields an equivalent set with at most one $\leqslant$-atom and at most one $\geqslant$-atom. ▮

Example 3.6 : Let ϕ_4 be $x \leqslant_1 y, x \leqslant_3 y, x \geqslant_4 y, y \geqslant_1 x$. We rewrite all gap atoms with x on the left of the predicate: $x \leqslant_1 y, x \leqslant_3 y, x \geqslant_4 y, x \leqslant_{-1} y$. Since $x \leqslant_3 y$ implies $x \leqslant_1 y$ and $x \leqslant_3 y$ implies $x \leqslant_{-1} y$, atoms $x \leqslant_1 y$ and $y \geqslant_1 x$ can be removed from ϕ_4. Thus, ϕ_4 is equivalent to $x \leqslant_3 y, x \geqslant_4 y$. ▮

We now consider how to translate constraints with more than two variables. For two gap atoms that share exactly one variable, we "join" their LTL translations using their shared variable.

Example 3.7 : Consider Rental from Example 3.5. Condition (ii): `Schedule`@y, `Compute`@z, $y \geqslant_5 z$, is translated by Gap as: $\texttt{Schedule} \wedge \mathsf{X}^5\mathsf{P}\,\texttt{Compute}$, and condition (iii) with atoms `Compute`@z, `Terminate`@w, $z \leqslant_0 w$ yields $\texttt{Compute} \wedge \mathsf{F}\,\texttt{Terminate}$. The shared variable z corresponds to the `Compute` proposition in both translations. Accordingly, the translations can be combined by placing the second formula in the position in the first formula where `Compute` appears, combining the duplicated proposition (bars added for illustration): $\psi = \overline{\texttt{Schedule} \wedge \mathsf{X}^5\mathsf{P}\,\underline{(\texttt{Compute}}}\underline{\;\wedge \mathsf{F}\,\texttt{Terminate})}$

To complete the translation of Rental, we join ψ with a LTL translation of condition (i). Recall that condition (i) covers atoms `Request`@x, `Schedule`@y, $x \leqslant_1 y$, $x \geqslant_{10} y$ and can be translated as:

$$\texttt{Request} \wedge \textstyle\bigvee_{1 \leqslant j \leqslant 10} \mathsf{X}^j\,\texttt{Schedule}$$

The atoms for conditions (ii) and (iii) and the atoms for condition (i) share the variable y, which corresponds with the `Schedule` proposition in both translations. To combine the translations, we place ψ at the position in the first formula where `Schedule` occurs, then remove the duplicated `Schedule` proposition:

$$\overline{\texttt{Request} \wedge \bigvee_{1\leqslant j\leqslant 10} \mathsf{X}^j \underline{(\texttt{Schedule}}}\underline{\wedge \mathsf{X}^5 \mathsf{P}\,(\texttt{Compute} \wedge \mathsf{F}\,\texttt{Terminate}))}$$ ▮

To generalize the joining technique in Example 3.7, we represent constraints with graphs. Recall that each constraint is a conjunctive formula with event atoms and gap atoms, i.e., unary and binary predicates on time variables. Thus, atoms can be faithfully represented by an undirected graph.

Definition : Let ϕ be a constraint. The *graph of* ϕ is an undirected, labeled graph $G_\phi = (V, E, L)$ such that V is the set of variables used in ϕ, E is the set of pairs (x, y) such that ϕ contains a gap atom using both x and y, and L is the mapping from $V \cup E$ such that

- for each variable $x \in V$, $L(x) = \{p \mid p@x \text{ is an event atom in } \phi\}$, and
- for each edge $(x, y) \in E$, $L(x, y) = \{\alpha \mid \alpha \text{ is a gap atom in } \phi \text{ using } x \text{ and } y\}$.

Furthermore, ϕ is *acyclic* if G_ϕ is acyclic, *connected* if G_ϕ is connected.

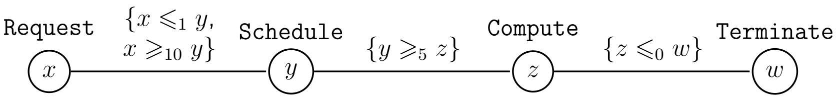

Figure 3.4: Graph of Rental

Example 3.8 : The graph of Rental (Fig. 3.4) has nodes $\{x, y, z, w\}$ labeled with `Request`, `Schedule`, `Compute`, `Terminate`, resp., edges $(x, y), (y, z), (z, w)$ labeled with $\{x \leqslant_1 y, x \geqslant_{10} y\}$, $\{y \geqslant_5 z\}$, and $\{z \leqslant_0 w\}$, resp., and is acyclic and connected. ▮

Definition : Let ϕ an acyclic, connected constraint with the graph $G_\phi = (V, E, L)$. For each node $x \in V$, the *derived tree of* ϕ *at* x, denoted T^x_ϕ, is the directed tree (V, E', L) rooted at x with nodes V, edges E' with the directed version of each edge in E pointing away from x, and the label mapping L. For each node $y \in V$, $T_\phi^x|_y$ denotes the subtree of T_ϕ^x rooted at y. For each node $z \in V$, let $Ch(z)$ denote z's children.

We formulate a translation of connected, acyclic constraints using their derived trees. Let ϕ be a connected, acyclic constraint and x a variable used in ϕ, where T_ϕ^x is the derived tree of ϕ at x. Intuitively, $\text{AcycConn}(T_\phi^x, x)$ denotes a translation of ϕ. In fact, for each node y in T_ϕ^x, the function $\text{AcycConn}(T_\phi^x, y)$ maps T_ϕ^x and y to an LTL formula, using the subtree $T_\phi^x|y$:

$$\text{AcycConn}(T_\phi^x, y) = \begin{cases} \bigwedge_{p \in L(y)} p & \text{if } y \text{ is a leaf} \\ \bigwedge_{p \in L(y)} p \wedge \bigwedge_{z \in Ch(y)} Gap_y(L(y,z))\text{AcycConn}(T_\phi^x, z) & \text{o.w.} \end{cases}$$

All event names in the label of y are used in the conjunction $\bigwedge_{p \in L(y)} p$ of LTL propositions. For each child z of y, the gap atoms $L(y,z)$ are translated to $\text{Gap}_{y,z}(L(y,z))$ and the algorithm makes a recursive call $\text{AcycConn}(T_\phi^x, z)$ to translate the subtree rooted at z with respect to z.

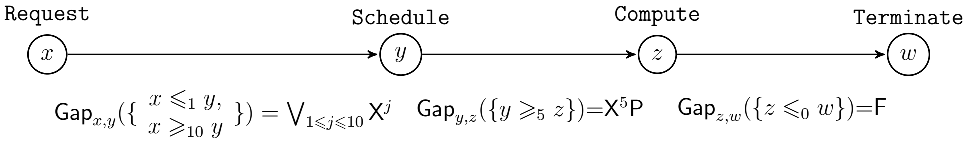
Request
Schedule
Compute
Terminate
x
y
z
w
$\mathsf{Gap}_{x,y}(\{ \begin{smallmatrix} x \leqslant_1 y, \\ x \geqslant_{10} y \end{smallmatrix} \}) = \bigvee_{1 \leqslant j \leqslant 10} \mathsf{X}^j$
$\mathsf{Gap}_{y,z}(\{y \geqslant_5 z\}) = \mathsf{X}^5\mathsf{P}$
$\mathsf{Gap}_{z,w}(\{z \leqslant_0 w\}) = \mathsf{F}$

Figure 3.5: The derived tree $T^x_{\textsc{Rental}}$ with Gap-mapping from edge labels to LTL operators

Example 3.9 : The AcycConn translation of Rental, abbreviated here as ϕ, is done using the derived tree T_ϕ^x shown in Fig. 3.5. Let ϕ_y, ϕ_z, ϕ_w be the subsets of ϕ such that the derived trees $T_{\phi_y}^y$, $T_{\phi_z}^z$,$T_{\phi_w}^w$ are the subtrees of T_ϕ^x rooted at y, z, w (resp.). We demonstrate the translation beginning with the leaf node w and moving towards the root x:

$\text{AcycConn}(T_{\phi_w}^w) = \texttt{Terminate}$

$\text{AcycConn}(T_{\phi_z}^z) = \texttt{Compute} \wedge \mathsf{F}\, \text{AcycConn}(T_{\phi_w}^w) = \texttt{Compute} \wedge \mathsf{F}\, \texttt{Terminate}$

$$\begin{aligned}\text{AcycConn}(T^y_\phi) &= \texttt{Schedule} \wedge \mathsf{X}^5\mathsf{P}\,\text{AcycConn}(T^z_{\phi_z})\\ &= \texttt{Schedule} \wedge \mathsf{X}^5\mathsf{P}\,(\texttt{Compute} \wedge \mathsf{F}\,\texttt{Terminate}), \quad \text{and}\\ \text{AcycConn}(T^x_{\phi_x}) &= \texttt{Request} \wedge \bigvee_{1 \leqslant j \leqslant 10} \mathsf{X}^j\ (\texttt{Schedule} \wedge \mathsf{X}^5\mathsf{P}(\texttt{Compute} \wedge \mathsf{F}\,\texttt{Terminate}))\end{aligned} \qquad \blacksquare$$

The following lemma relates two notions of satisfaction defined in Section 3.2: the satisfaction of a constraint by an enactment and an assignment, and the satisfaction of an LTL formula by a trace at an instant.

Let ϕ be an acyclic, connected constraint where $x, y \in \mathit{var}(\phi)$. Let $\mathit{atoms}(T^x_\phi)$ ($\mathit{atoms}(T^x_\phi|_y)$) denote the set of atoms α in ϕ such that T^x_ϕ (resp., $T^x_\phi|_y$) is the tree derived from α.

Lemma 3.2 : Let η be an enactment, ϕ a connected, acyclic constraint, x, y variables in $\mathit{var}(\phi)$, and $i \in \mathbb{N}$ a timestamp. The following statements are equivalent:

1. There is an assignment σ such that $\sigma(y) = i$ and $\eta \models \mathit{atoms}(T^x_\phi|_y)[\sigma]$,
2. $\eta, i \models \text{AcycConn}(T^x_\phi, y)$.

Proof: The proof is accomplished by mathematical induction on the height n of the subtree $T^x_\phi|_y$. Without loss of generality, let i be a timestamp and L the label function of T^x_ϕ.

Base case: y is a leaf in T^x_ϕ.
Since ϕ is connected, T^x_ϕ is connected, and $T^x_\phi|_y$ has just the node y. The atoms for $T^x_\phi|_y$ are the event atoms in ϕ that use y, i.e., $\mathit{atoms}(T^x_\phi|_y) = \{\, p@y \mid p \in L(y) \,\}$. The AcycConn translation of T^x_ϕ w.r.t. y is $\text{AcycConn}(T^x_\phi, y) = \bigwedge_{p \in L(y)} p$. To establish the base case, it suffices to show that

(a) there is an assignment σ such that $\sigma(y) = i$ and $\eta \models \{p@y \mid p \in L(y)\}[\sigma]$
iff (b) $\eta, i \models \bigwedge_{p \in L(y)} p$.

To show (a) implies (b), we assume for some assignment σ, $\sigma(y) = i$ and $\eta \models \{p@y \mid p \in L(y)\}[\sigma]$. Isolating the satisfaction of each event atom, we have: for each event name p in

$L(y)$, $\sigma(y) \in \eta(p)$. Under the mapping between enactments and traces (see Section 3.2), this is equivalent to: for each $p \in L(y)$, $\eta, \sigma(y) \models p$. Combining the event names associated with y with conjunction, we have: $\eta, \sigma(y) \models \bigwedge_{p \in L(y)} p$. Since $\sigma(y) = i$, we have $\eta, i \models \bigwedge_{p \in L(y)} p$.

To show (b) implies (a), we assume $\eta, i \models \bigwedge_{p \in L(y)} p$. It follows that for each $p \in L(y)$, $\eta, i \models p$. Using again the mapping between enactments and traces, this is equivalent to: for each $p \in L(y)$, $i \in \eta(p)$. Let σ be an assignment such that $\sigma(y) = i$. For each $p \in L(y)$, $\sigma(y) \in \eta(p)$, i.e., for each $p \in L(y)$, $\eta \models \{p@y\}[\sigma]$. Therefore, $\eta \models \{p@y \mid p \in L(y)\}[\sigma]$.

Induction hypothesis: For each node $z \in Ch(y)$ in T^x_ϕ, i.e., each subtree of height at most $n-1$, for each timestamp $j \in \mathbb{N}$, the following statements are equivalent:

(1z) There is an assignment σ_z such that $\sigma_z(z) = j$ and $\eta \models atoms(T^x_\phi|_z)[\sigma_z]$.

(2z) $\eta, j \models \text{AcycConn}(T^x_\phi, z)$.

Induction step: Let the subtree $T^x_\phi|_y$ have height n. We shall show that

(A) There is an assignment σ such that $\sigma(y) = i$ and $\eta \models atoms(T^x_\phi|_y)[\sigma]$

iff (B) $\eta, i \models \text{AcycConn}(T^x_\phi, y)$.

First, we show that (A) implies (B). We begin by assuming there is an assignment σ such that $\sigma(y) = i$ and $\eta \models atoms(T^x_\phi|_y)[\sigma]$.

Claim A: For each child z of y, $\eta, i \models \text{Gap}_{y,z}(L(y,z))\text{AcycConn}(T^x_\phi, z)$.

We prove Claim A with three cases of the $L(y,z)$. Since $T^x_\phi|_z$ is a subtree of $T^x_\phi|_y$ we have $\eta \models atoms(T^x_\phi|_z)[\sigma]$. Letting $j = \sigma(z)$, (1z) of the inductive hypothesis holds with the assignment σ_z. Applying the inductive hypothesis, we have $\eta, j \models \text{AcycConn}(T^x_\phi, z)$. It remains to establish (B): $\eta, i \models \text{AcycConn}(T^x_\phi, y)$.

Since z and y are nodes in $T^x_\phi|_y$ and $\eta \models atoms(T^x_\phi|_y)[\sigma]$, we immediately have $\eta \models L(y,z)[\sigma]$. By Lemma 3.1, $L(y,z)$ is equivalent to one of the following three constraints.

<u>Case 1</u>: $L(y,z) \equiv \{y \leqslant_m z\}$ for some $m \in \mathbb{Z}$.

Since $\eta \models L(y,z)[\sigma]$, we have $\eta \models (y \leqslant_m z)[\sigma]$, i.e., $\sigma(y){+}m \leqslant \sigma(z)$. Recall that $\sigma(y)=i$ and $\sigma(z)=j$. We have $i{+}m{+}k=j$ for some $k\in\mathbb{N}$. Replacing j in the inductive hypothesis, we obtain $\eta, i{+}m{+}k \models \text{AcycConn}(T^x_\phi, z)$. It means that $\text{AcycConn}(T^x_\phi, z)$ is satisfied by η at some future time of time $i{+}m$, i.e., $\eta, i{+}m \models \mathsf{F}\text{AcycConn}(T^x_\phi, z)$. Adjusting the index of satisfaction by prepending m next operators, we have $\eta, i \models \mathsf{X}^m\mathsf{F}\text{AcycConn}(T^x_\phi, z)$.

Case 2: $L(y,z) \equiv \{y \geqslant_m z\}$ for some $m\in\mathbb{Z}$.

Similarly, $\eta \models L(y,z)[\sigma]$ implies $\eta \models (y \geqslant_m z)[\sigma]$, and $\sigma(y){+}m \geqslant \sigma(z)$. Since $\sigma(y)=i$ and $\sigma(z)=j$, we have $i{+}m{-}k=j$ for some $k\in\mathbb{N}$. From the inductive hypothesis, we have $\eta, i{+}m{-}k \models \text{AcycConn}(T^x_\phi, z)$. It is easy to see η satisfies $\mathsf{P}\text{AcycConn}(T^x_\phi, z)$ at $i{+}m$ and prepending m next operators, we have $\eta, i \models \mathsf{X}^m\mathsf{P}\text{AcycConn}(T^x_\phi, z)$.

Case 3: $L(y,z) \equiv \{y \leqslant_m z, y \geqslant_{m'} z\}$ for some $m,m'\in\mathbb{Z}$.

As before, $\eta \models L(y,z)[\sigma]$ implies $\eta \models (y{\leqslant_m}z) \wedge (y{\geqslant_{m'}}z)[\sigma]$, and $\sigma(y){+}k=\sigma(z)$ for some $m \leqslant k \leqslant m'$. Since $\sigma(y)=y$ and $\sigma(z)=j$, $i{+}k=j$. From the inductive hypothesis, it follows that $\eta, i{+}k \models \text{AcycConn}(T^x_\phi, z)$. Since k is constrained between m, m' we have $\eta, i \models \bigvee_{m\leqslant k\leqslant m'} \mathsf{X}^k\ \text{AcycConn}(T^x_\phi, z)$.

By Lemma 3.1 and the above cases, Claim A is proved.

Claim A implies that (C): $\eta, i \models \bigwedge_{z\in Ch(y)} \text{Gap}_{y,z}(L(y,z))\text{AcycConn}(T^x_\phi, z)$.

Earlier we assumed (A) was true, so $\eta \models \alpha[\sigma]$ for each event atom α that uses y. It follows that $\eta \models \{p@y \mid p\in L(y)\}[\sigma]$. Using a similar reasoning to that in the base case, we have (D) $\eta, i \models \bigwedge_{p\in L(y)} p$. Combining (C) and (D), we have $\eta, i \models (\bigwedge_{p\in L(y)} p) \wedge \bigwedge_{z\in Ch(y)} \text{Gap}_{y,z}(L(y,z))\text{AcycConn}(T^x_\phi, z)$, i.e., $\eta, i \models \text{AcycConn}(T^x_\phi, y)$ (applying the definition of AcycConn).

Now, we show that (B) implies (A) for the inductive step. First we assume (B): $\eta, i \models \text{AcycConn}(T^x_\phi, y)$. Expanding AcycConn with its definition, it follows that $\eta, i \models (\bigwedge_{p\in L(y)} p) \bigwedge_{z\in Ch(y)} \text{Gap}_{y,z}(L(y,z))\text{AcycConn}(T^x_\phi, z)$.

We let z be an arbitrary child of y. Then, $\eta, i \models \mathrm{Gap}_{y,z}(L(y,z))\mathrm{AcycConn}(T^x_\phi, z)$ holds.

Claim B: There is some timestamp j such that

(c1) $\eta, j \models \mathrm{AcycConn}(T^x_\phi, z)$, and

(c2) for each assignment σ' such that $\sigma'(y) = i$ and $\sigma'(z) = j$, $\eta \models L(y,z)[\sigma']$.

We establish Claim B with the following three cases of $L(y,z)$.

Case 1: $L(y,z) \equiv \{y \leqslant_m z\}$ for some $m \in \mathbb{Z}$.
By definition, $\mathrm{Gap}_{y,z}(L(y,z)) = \mathsf{X}^m\mathsf{F}$, and we have $\eta, i \models \mathsf{X}^m\mathsf{F}\mathrm{AcycConn}(T^x_\phi, z)$. The latter is equivalent to η satisfying $\mathsf{F}\mathrm{AcycConn}(T^x_\phi, z)$ at time $i{+}m$. It follows that $\eta, i{+}m{+}k \models \mathrm{AcycConn}(T^x_\phi, z)$ for some $k \in \mathbb{N}$. For $j = i{+}m{+}k$, $\eta, j \models \mathrm{AcycConn}(T^x_\phi, z)$. Since k is nonnegative and $L(y,z) \equiv \{y \leqslant_m z\}$, $\eta \models L(y,z)[\sigma']$ for all assignments σ' such that $\sigma'(y) = i$ and $\sigma'(z) = j$.

Case 2: $L(y,z) \equiv \{y \geqslant_m z\}$ for some $m \in \mathbb{Z}$.
In this case, $\mathrm{Gap}_{y,z}(L(y,z)) = \mathsf{X}^m\mathsf{P}$, and $\eta, i \models \mathsf{X}^m\mathsf{P}\mathrm{AcycConn}(T^x_\phi, z)$. By LTL semantics, $\eta, i{+}m \models \mathsf{P}\mathrm{AcycConn}(T^x_\phi, z)$, and $\eta, i{+}m{-}k \models \mathrm{AcycConn}(T^x_\phi, z)$ for some $k \in \mathbb{N}$. Letting $j = i{+}m{-}k$, $\eta, j \models \mathrm{AcycConn}(T^x_\phi, z)$, and $i \geqslant_m j$ (k is non-negative). Since $L(y,z) \equiv \{y \geqslant_m z\}$, we conclude that $\eta \models L(y,z)[\sigma']$ for all assignments σ' such that $\sigma'(y) = i$ and $\sigma'(z) = j$.

Case 3: $L(y,z) \equiv \{y \leqslant_m z, y \geqslant_{m'} z\}$ for some $m, m' \in \mathbb{Z}$.
Note that $m \leqslant m'$ must hold, otherwise the constraint $L(y,z)$ is unsatisfiable. By definition, $\mathrm{Gap}_{y,z}(L(y,z)) = \bigvee_{m \leqslant k \leqslant m'} \mathsf{X}^k$, and $\eta, i \models \bigvee_{m \leqslant k \leqslant m'} \mathsf{X}^k\mathrm{AcycConn}(T^x_\phi, z)$. It follows that for some $k \in [m..m']$, $\eta, i \models \mathsf{X}^k\mathrm{AcycConn}(T^x_\phi, z)$, and $\eta, i{+}k \models \mathrm{AcycConn}(T^x_\phi, z)$. Let $j = i{+}k$. We have $i \leqslant_m j \wedge i \geqslant_{m'} j$ and $\eta, j \models \mathrm{AcycConn}(T^x_\phi, z)$. Since $L(y,z) \equiv \{y \leqslant_m z, y \geqslant_{m'} z\}$, $\eta \models L(y,z)[\sigma']$ for all assignments σ' such that $\sigma'(y) = i$ and $\sigma'(z) = j$.

This concludes the proof of Claim B.

We now prove (B) implies (A). From (B), we have

$\eta, i \models \bigwedge_{z \in Ch(y)} \text{Gap}_{y,z}(L(y,z))\text{AcycConn}(T^x_\phi, z)$. For each child $z \in Ch(y)$, we have $\eta, i \models \text{Gap}_{y,z}(L(y,z))\text{AcycConn}(T^x_\phi, z)$. Applying Claim B, there is a timestamp j such that $\eta, j \models \text{AcycConn}(T^x_\phi, z)$. By the inductive hypothesis, it follows that (1z) also holds: there is some assignment σ_z such that $\sigma_z(z) = j$ and $\eta \models atoms(T^x_\phi|_z)[\sigma_z]$.

Since $T^x_\phi|_y$ is a tree with subtrees $T^x_\phi|_z$'s where $z \in Ch(y)$, we can combine all assignments σ_z's for y's children $z \in Ch(y)$ into an assignment σ such that $\sigma(y) = i$. Note that for each child $z \in Ch(y)$ and each node v in subtree $T^x_\phi|_z$, $\sigma(v) = \sigma_z(v)$.

By (B), $\eta, i \models \text{AcycConn}(T^x_\phi, y)$, where

$\text{AcycConn}(T^x_\phi, y) = \ (\bigwedge_{p \in L(y)} p) \wedge \bigwedge_{z \in Ch(y)} \text{Gap}_{y,z}(L(y,z))$.

To complete the proof of the lemma, we show $\eta \models atoms(T^x_\phi|_y)[\sigma]$. Consider an arbitrary atom α in $atoms(T^x_\phi|_y)$.

<u>Case 1</u>: α is an event atom using y. From identical reasoning as given in the base case, $\eta \models \{p@y | p \in L(y)\}[\sigma]$. Since $\sigma(y) = i$, $\eta \models \alpha[\sigma]$.

<u>Case 2</u>: α is a gap atom in the label $L(y,z)$ for some $z \in Ch(y)$. By construction of σ, $\sigma(y) = i$ and $\sigma(z) = \sigma_z(z) = j$, where j is obtained from Claim B σ_z from applying the inductive hypothesis with z. Thus, we have $\eta \models \alpha[\sigma]$.

<u>Case 3</u>: α is an event atom or gap atom in $atoms(T^x_\phi|_z)$ for some $z \in Ch(y)$. By the inductive hypothesis with z we have an assignment σ_z such that $\eta \models atoms(T^x_\phi|_z)[\sigma_z]$. From the construction of σ, σ and σ_z agree on all variables v used in $atoms(T^x_\phi|_z)$. Thus, we have $\eta \models atoms(T^x_\phi|_z)[\sigma]$.

These cases cover all atoms in $atoms(T^x_\phi|_y)$. Therefore, $\eta \models atoms(T^x_\phi|_y)[\sigma]$. This establishes (A) is established.

This concludes the proof of Lemma 3.2. ▮

A key observation in proving Lemma 3.2 is that an assignment satisfying a constraint for a given enactment identifies the instants where subformula of AcycConn, especially propositions, are true in the enactment's trace. Note that $atoms(T^x_\phi|_x) = atoms(T^x_\phi) = \phi$

for each constraint ϕ and each $x \in \mathit{var}(\phi)$. Then, we have the following corollary.

Corollary 3.3 : Let η be an enactment, ϕ a connected, acyclic constraint, x a variable in $\mathit{var}(\phi)$, and $i \in \mathbb{N}$ a timestamp. The following are equivalent:

1. There is an assignment σ, such that $\sigma(x) = i$ and $\eta \models \phi[\sigma]$.
2. $\eta, i \models \mathrm{AcycConn}(T^x_\phi, x)$.

We now use AcycConn to define a function to translate acyclic and possibly disconnected constraints. Let ϕ be an arbitrary acyclic constraint; ϕ can be partitioned into k connected constraints $\phi_1, ..., \phi_k$. Let x_j be a variable in $\mathit{var}(\phi_j)$ for each $1 \leqslant j \leqslant k$; a translation of each ϕ_j is given by $\mathrm{AcycConn}(T^{x_j}_{\phi_j}, x_j)$. For each pair $1 \leqslant j, l \leqslant k$, if $j \neq l$ the sets $\mathit{var}(\phi_j)$ and $\mathit{var}(\phi_k)$ are disjoint. Thus, assignments to these variables, and the instants to satisfy $\mathrm{AcycConn}(T^{x_j}_{\phi_j}, x_j)$ and $\mathrm{AcycConn}(T^{x_l}_{\phi_l}, x_1)$ are independent. Accordingly, we combine these translations by choosing one variable, here x_1, as an "anchor", translating ϕ_1 as $\mathrm{AcycConn}(T^{x_1}_{\phi_1}, x_1)$, then conjuncting this formula with the AcycConn translations of the other connected constraints, offset by past and future operators.

The translation of ϕ anchored at x_1 is as follows: $\mathrm{Acyc}(T^{x_1}_\phi, x_1) =$

$$\mathrm{AcycConn}(T^{x_1}_{\phi_1}, x_1) \wedge \bigwedge_{2 \leqslant j \leqslant k} \Big(\mathsf{P}\mathrm{AcycConn}(T^{x_j}_{\phi_j}, x_j) \vee \mathsf{F}\mathrm{AcycConn}(T^{x_j}_{\phi_j}, x_j) \Big)$$

The translation $\mathrm{Acyc}(T^{x_1}_\phi, x_1)$ is anchored at x_1 because if $\eta \models \phi[\sigma]$, the LTL formula $\mathrm{Acyc}(T^{x_1}_\phi, x_1)$ is true at instant $\sigma(x_1)$ in η. This anchoring is crucial in connecting the satisfaction of constraints and LTL formulas.

The following lemma extends Lemma 3.2 to arbitrary acyclic constraints.

Lemma 3.4 : Let η be an enactment, ϕ an acyclic constraint, x a variable in ϕ, and $i \in \mathbb{N}$ a timestamp. The following are equivalent:

1. There is an assignment σ such that $\eta \models \phi[\sigma]$ and $\sigma(x) = i$.

2. $\eta, i \models \text{Acyc}(T^x_\phi, x)$.

Proof: Let i be a timestamp and $\phi_1, ..., \phi_k$ be connected constraints that partition ϕ. Let x_j be a variable in ϕ_j for each $1 \leqslant j \leqslant k$. Without loss of generality, assume $x = x_1$.

Assume that for some assignment σ, $\eta \models (\phi_1 \cup \cdots \cup \phi_k)[\sigma]$ and $\sigma(x_1) = i_1$. Then, $\eta \models \phi_1[\sigma]$ and $\sigma(x_1) = i$. Applying Lemma 3.2, with η, ϕ_1, x_1, and i_1, yields $\eta, i_1 \models \text{AcycConn}(T^{x_1}_{\phi_1}, x_1)$. Similarly, for each $2 \leqslant j \leqslant k$, $\eta \models \phi_j[\sigma]$ and for some $i_j \in \mathbb{N}$, $\sigma(x_j) = i_j$. Applying Lemma 3.2 with η, ϕ_j, x_j and i_j yields $\eta, i_j \models \text{AcycConn}(T^{x_j}_{\phi_j}, x_j)$. Because *var*$(\phi_1)$ and *var*(ϕ_j) are disjoint for each $j \neq 1$, the instants i_1 and i_j are independent. Thus, the values $i_2, ..., i_k$ are arbitrarily ordered with respect to i_1 (and each other), so $\eta, i_1 \models \mathsf{P}\text{AcycConn}(T^{x_j}_{\phi_j}, x_j) \vee \mathsf{F}\text{AcycConn}(T^{x_j}_{\phi_j}, x_j)$ for each $2 \leqslant j \leqslant k$. Combining these with conjunction and $\eta, i_1 \models \text{AcycConn}(T^{x_1}_{\phi_1}, x_1)$ yields $\eta, i \models \text{Acyc}(T^x_\phi, x)$.

For the converse, assume $\eta, i \models \text{Acyc}(T^x_\phi, x)$. By the definition of Acyc, we have $\eta, i \models \text{AcycConn}(T^{x_1}_{\phi_1}, x_1) \wedge \bigwedge_{2 \leqslant j \leqslant k} \Big(\mathsf{P}\text{AcycConn}(T^{x_j}_{\phi_j}, x_j) \vee \mathsf{F}\text{AcycConn}(T^{x_j}_{\phi_j}, x_j)\Big)$. Then, $\eta, i \models \text{AcycConn}(T^{x_1}_{\phi_1}, x_1)$ and by the semantics of the P and F operators, for some $i_2, ..., i_k$, $\eta, i_j \models \text{AcycConn}(T^{x_j}_{\phi_j}, x_j)$. Applying Lemma 3.2 to the above statement for η, ϕ_1, x_1, and i yields $\eta \models \phi_1[\sigma_1]$ and $\sigma_1(x_1) = i$ for some assignment σ_1 and for ϕ_i, x_j, and i_j for $2 \leqslant j \leqslant k$, Lemma 3.2 yields $\eta, i_j \models \eta \models \phi_i[\sigma_i]$ for some assignment σ_i. Because $\phi_1, \ldots, \phi_k$ are pairwise disconnected, *var*$(\phi_1), \ldots,$ *var*(ϕ_k) are pairwise disjoint. Then there is an assignment σ such that $\sigma(v) = \sigma_j(v)$ for each $v \in$ *var*(ϕ_j). Because $\phi = \phi_1 \cup \cdots \cup \phi_k$, it follows that $\eta \models \phi[\sigma]$. ▮

Example 3.10 : Consider a constraint PAID $= \{\texttt{Request}@x$, $\texttt{Schedule}@y$, $\texttt{Payment}@z$, $x \leqslant_3 y\}$ that selects a trio of timestamps to satisfy one gap atom. Note that PAID is acyclic, not connected, and can be partitioned into connected constraints:

$\psi_1 = \{\texttt{Request}@x$, $\texttt{Schedule}@y, x \leqslant_3 y\}$ and $\psi_2 = \{\texttt{Payment}@z\}$. For these two constraints, $\text{AcycConn}(T^x_{\psi_1}, x) = \texttt{Request} \wedge \mathsf{X}^3\mathsf{F}\ \texttt{Schedule}$ and $\text{AcycConn}(T^z_{\psi_2}, z) = \texttt{Payment}$. Picking x to anchor the translation yields:

$$\begin{aligned}\mathrm{Acyc}(T^{x}_{\text{PAID}}, x) &= \mathrm{AcycConn}(T^{x}_{\psi_1}, x)\\ &\qquad\qquad \wedge\Big(\mathsf{P}\,\mathrm{AcycConn}(T^{z}_{\psi_2}, z) \vee \mathsf{F}\,\mathrm{AcycConn}(T^{z}_{\psi_2}, z)\Big) \qquad\qquad \blacksquare\\ &= (\texttt{Request} \wedge \mathsf{X}^3\mathsf{F}\ \texttt{Schedule}) \wedge (\mathsf{P}\ \texttt{Payment} \vee \mathsf{F}\ \texttt{Payment})\end{aligned}$$

We now turn to the main technical result of the chapter: translating singly-linked, acyclic rules.

A dataless rule $\phi \rightarrow \psi$ is *singly-linked* if ϕ and ψ share at most one variable. Recall that an enactment η satisfies a rule $\phi \rightarrow \psi$ if for every assignment σ where $\eta \models \phi[\sigma]$, there is some assignment σ' that extends σ such that $\eta \models \psi[\sigma']$. When ϕ and ψ share one variable, the key idea in translating rules is joining the LTL translations of ϕ and ψ at an instant corresponding to their common variable. The following example illustrates this idea.

Example 3.11 : Consider the constraint RENTAL in Example 3.5 and a constraint BILLING $= \{\texttt{Payment}@u, \texttt{Receipt}@v, u \leqslant_0 v, y \geqslant_7 u\}$. It is satisfied when a `Payment` event is simultaneous with or followed by a `Receipt`, and the `Payment` instance is no later than some time v+7. Note the timestamps in the BILLING constraint are limited by the time `Schedule` occurs in the RENTAL constraint.

The rule `TimelyPayment` is expressed as: RENTAL → BILLING. This rule is singly-linked and acyclic because the union of its constraints is acyclic and y is the only variable the constraints share. Note that RENTAL is closed, but BILLING is not. However, BILLING$' =$ BILLING $\cup \{\texttt{Schedule}@y\}$ is closed. In Fig. 3.6, the graphs of RENTAL and of BILLING$'$ are shown, with a dotted line indicating that BILLING$'$ has been extended with the event atom from the shared variable y.

We translate RENTAL and BILLING$'$ using the Acyc function anchored at y:

$$\mathrm{Acyc}(T^{y}_{\text{RENTAL}}, y) = \texttt{Schedule} \wedge ((\bigvee_{-10\leqslant j\leqslant -1} \mathsf{X}^j\ \texttt{Request}) \wedge \mathsf{X}^5\mathsf{P}(\texttt{Compute} \wedge \mathsf{F}\ \texttt{Terminate}))$$

$$\mathrm{Acyc}(T^{y}_{\text{BILLING}'}, y) = \texttt{Schedule} \wedge \mathsf{X}^7\mathsf{P}(\texttt{Payment} \wedge \mathsf{F}\ \texttt{Receipt})$$

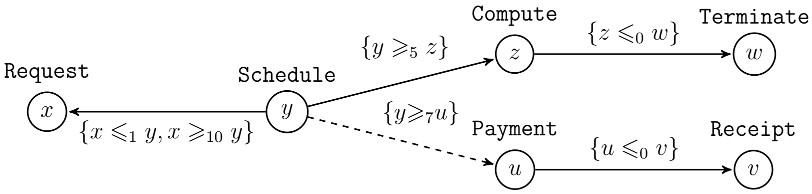

Figure 3.6: The tree for `TimelyPayment` built from trees for Rental and Billing, rooted at y

`TimelyPayment` expresses the requirement that each assignment that assigns y the instant i and satisfies Rental for an enactment can be extended to satisfy Billing′. Using Lemma 3.4, this is equivalent to requiring that each instant i in the enactment's trace that satisfies $\text{Acyc}(T^y_{\text{Rental}}, y)$ also satisfies $\text{Acyc}(T^y_{\text{Billing}'}, y)$. We use implication to reflect this requirement with respect to the instant i. Because y can be assigned an arbitrary timestamp, we place the implication in the scope of an LTL global operator. The translation of `TimelyPayment` is $\mathsf{G}(\text{Acyc}(T^y_{\text{Rental}}, y) \rightarrow \text{Acyc}(T^y_{\text{Billing}'}, y))$ ∎

The translation in Example 3.11 generalizes to a translation function SingAcycToLTL for singly-linked, acyclic rules, given below:

SingAcycToLTL$(\phi \rightarrow \psi)$ =

$$\begin{cases} \mathsf{G}(\text{Acyc}(T^x_\phi, x) \rightarrow \text{Acyc}(T^x_{\psi'}, x)) & \text{If } \phi \text{ and } \psi \text{ share variable } x \\ \mathsf{G}(\text{Acyc}(T^x_\phi, x) \rightarrow \big(\mathsf{P}\text{Acyc}(T^z_\psi, z) \vee \mathsf{F}\text{Acyc}(T^z_\psi, z)\big)) & \\ & \text{Otherwise, with } z \in \mathit{var}(\psi) \end{cases}$$

Let $\phi \rightarrow \psi$ be a singly-linked, acyclic rule. In the first case, ϕ and ψ share a variable x. Let ψ' be the union of ψ and the set of event atoms in ϕ with x. Note that ψ' is closed. We obtain translations $\text{Acyc}(T^x_\phi, x)$ and $\text{Acyc}(T^x_{\psi'}, x)$ of ϕ and ψ' (resp.) anchored at x. Then SingAcycToLTL$(\phi \rightarrow \psi)$ is $\mathsf{G}(\text{Acyc}(T^x_\phi, x) \rightarrow \text{Acyc}(T^x_{\psi'}, x))$ as shown in the above.

When ϕ and ψ have no common variables, the lack of a shared variable means the instant(s) satisfying $\text{Acyc}(T^z_\psi, z)$ is independent of the instant(s) satisfying $\text{Acyc}(T^x_\phi, x)$, so $\text{Acyc}(T^z_\psi, z)$ can be satisfied anywhere in the trace relative to $\text{Acyc}(T^x_\phi, x)$, resulting in the second formula above.

The following theorem establishes key properties of SingAcycToLTL.

Theorem 3.5 : Let η be an enactment, $\phi \to \psi$ a singly-linked, acyclic rule. The following are equivalent:

1. $\eta \models \phi \to \psi$

2. $\eta, 0 \models \text{SingAcycToLTL}(\phi \to \psi)$

and $|\text{SingAcycToLTL}(\phi \to \psi)|$ is $O(2^{|\phi \to \psi|^2})$.

Also, for a set of rules R, the following statements are equivalent:

3. $\eta \models R$

4. $\eta, 0 \models \bigwedge_{r \in R} \text{SingAcycToLTL}(r)$

Proof: First, we show that (1) implies (2). Assume $\eta \models \phi \to \psi$. We consider two cases for the variables in ϕ and ψ.

Case 1: ϕ and ψ share some variable x.

From the definition, $\text{SingAcycToLTL}(\phi \to \psi) = \mathsf{G}(\text{Acyc}(T^x_\phi, x) \to \text{Acyc}(T^x_\psi, x))$. To show $\eta, 0 \models \text{SingAcycToLTL}(\phi \to \psi)$ it is sufficient to show that for every instant i of the trace, if $\eta, i \models \text{Acyc}(T^x_\phi, x)$, then $\eta, i \models \text{Acyc}(T^x_\psi, x)$.

Let i be an arbitrary instant of η such that $\eta, i \models \text{Acyc}(T^x_\phi, x)$. By Lemma 3.4, there is an assignment σ such $\eta \models \phi[\sigma]$ and $\sigma(x) = i$. By assumption $\eta \models \phi \to \psi$, there is some assignment σ' that extends σ (i.e., $\sigma'(x) = i$) such that $\eta \models \psi[\sigma']$. Applying Lemma 3.4 again to $\eta \models \psi[\sigma']$ and $\sigma'(x) = i$, yields $\eta, i \models \text{Acyc}(\psi, x)$.

Case 2: ϕ and ψ share no variables.

Let $x \in \mathit{var}(\phi)$ and $z \in \mathit{var}(\psi)$. Applying SingAcycToLTL, it suffices to show that for each instant i where $\text{Acyc}(T^x_\phi, x)$ is satisfied, $\mathsf{P}\text{Acyc}(T^z_\psi, z) \vee \mathsf{F}\text{Acyc}(T^z_\psi, z)$ is also satisfied at i, i.e., $\text{Acyc}(T^z_\psi, z)$ is satisfied somewhere in the trace. Let $i \in \mathbb{N}$. Assume $\eta, i \models \text{Acyc}(T^x_\phi, x)$. By Lemma 3.4, there is an assignment σ such that $\eta \models \phi[\sigma]$ and

$\sigma(x) = i$. From the assumption that $\eta \models \phi \to \psi$, there is an assignment σ' such that $\eta \models \psi[\sigma']$ and $\sigma'(z) = j$. Applying Lemma 3.4 again, we have $\eta, j \models \mathrm{Acyc}_{\psi,z}$. Either $i \geqslant j$ and thus $\eta, i \models \mathsf{P}\mathrm{Acyc}_{\psi,z}$, or $i \leqslant j$ and $\eta, i \models \mathsf{F}\mathrm{Acyc}(T^z_\psi, z)$. In either case, the needed condition is established.

The converse direction is proved similarly.

For the second equivalence between (3)and (4), note that a set of rules is interpreted as a conjunction, matching the outermost LTL conjunction in (4).

Size of Translation: Let $\gamma(\phi \to \psi)$ be the size of SingAcycToLTL$(\phi \to \psi)$. We first address the size of AcycConn for ϕ and ψ, which is used in Acyc and SingAcycToLTL. The tree T^x_ϕ used in AcycConn rooted at an arbitrary $x \in \mathit{var}(\phi)$. Let y be a child of x. By the definition of AcycConn and its function Gap, AcycConn(T^x_ϕ, x) contains AcycConn(T^x_ϕ, y) duplicated $m - n + 1$ times if $\{x + n \leqslant y, x + m \geqslant y\}$ is in ϕ for some gaps n, m and duplicated only once if ϕ has just one gap for x and y. Then, the resulting LTL formulas are larger wrt the gaps for x and y when ϕ contains an 0-gap lower bound and m-gap upper bound between x and y. Furthermore, because AcycConn is recursive, the resulting LTL formulas are larger when T^x_ϕ has a larger height rather than a larger breadth. Thus, the largest output of AcycConn(T^x_ϕ, x) is when T^x_ϕ is a path graph with root x, where for some positive m, every edge label is $\{x+0 \leqslant y, x+m \geqslant y\}$. When $|\mathit{var}(\phi)| = v$, the size $U(v)$ of AcycConn(T^x_ϕ, x) is given by the recurrence relation: $U(v) = 1+1+(m-1)+\frac{m(m+1)}{2}+(m+1)\cdot U(v-1)$, with one proposition for the event atom, one conjunction operator, $m - 1$ disjunction operators, $\frac{m}{2}(m + 1)$ next LTL operators, and $(m+1)$ copies of $U(v - 1)$ (the recursive call). Solving this recurrence has a closed form:

$$U(v) = \sum_{i=1}^{v-1} \prod_{j=1}^{i-1} (m+1) + \sum_{i=1}^{v-1} (\prod_{j=1}^{i-1} (m+1)) \frac{m}{2}(m+1) = (\frac{m}{2}(m+1)+1) \sum_{i=1}^{v-1} (m+1)^{i+1}$$

which simplifies to $U(v) = (\frac{m}{2}(m + 1) + 1)(\frac{(m+1)^v-(m+1)}{m^2+m})$. When m is represented in the input $|\phi|$ in binary, $m \leqslant 2^{|\phi|}$. Using $m \leqslant 2^{|\phi|}$ and $v \leqslant |\phi|$, we have $U(v) \leqslant (\frac{(2^{|\phi|})}{2}((2^{|\phi|}) + 3) + 1)(\frac{((2^{|\phi|})+1)^{|\phi|} - ((2^{|\phi|})+1)}{(2^{|\phi|})^2 + (2^{|\phi|})})$. Then, $U(v)$ and $|\mathrm{AcycConn}(T_\phi^x, x)|$ are $O(2^{|\phi|^2})$.

The growth of γ follows directly from the size of AcycConn. The path graph T_ϕ is connected, so $|Acyc(T_\phi^x, x)|$ is $O(|AcycConn(T_\phi^x|))$. Then, whether or not ϕ and ψ share a variable, $\gamma(\phi \to \psi)$ is $O(|Acyc(T_\phi^x, x)| + |Acyc(T_\psi^z, z)|)$ for some $z \in \mathit{var}(\psi)$. It follows that that $\gamma(\phi \to \psi)$ is $O(2^{|\phi|^2} + 2^{|\psi|^2})$. Given that $2^{|\phi|^2} + 2^{|\psi|^2} \leqslant 2^{|\phi|^2} \cdot 2^{|\psi|^2}$ and $|\phi|^2 + |\psi|^2 \leqslant (|\phi| + |\psi|)^2$, it follows that $\gamma(\phi \to \psi)$ is $O(2^{|\phi \to \psi|^2})$. Thus, this translation for singly-linked, acyclic rules is single-exponential in the size of the rule. ❚

Proving (1) and (2) are equivalent in Theorem 3.5 relies on the construction in the proof of Lemma 3.4, which connects a satisfying assignment for a constraint with one instant in the trace when the constraint's translation is satisfied. This is achieved by observing that variables' assigned values correspond to instants where propositions for associated event names and the subformulas of the constraint's translation are satisfied. Also note that joining constraints using their shared variable ensures assignments to the left-hand side of a rule are extended by assignments to its right-hand side, that assign the same timestamp to the shared variable.

3.4 All-Order Translation of Singly-Linked Rules

Now we present a more general translation function, which translates singly-linked, but not necessarily acyclic, rules. This removes the acyclicity requirement found in the previous section, though the space complexity of the LTL formula increases. To achieve this, we decompose constraints into an equivalent disjunction of "primitive" constraints, and translate each primitive constraint. Finally, we note that this second translation algorithm that doesn't produce past-time LTL operators for a subset of singly-linked rules.

Let V be a finite set of variables. An *enumeration* of V is an ordered sequence without repetition of all elements of V.

Definition : Let ϕ be a constraint, $x_1, ..., x_n$ an enumeration s of $var(\phi)$. A constraint p is *primitive for ϕ with respect to s* if (i) $var(p) = var(\phi)$, (ii) p and ϕ have the same event atoms, and (iii) for each $i \in [1..(n-1)]$, p contains exactly one formula of form $x_i + b = x_{i+1}$ for an integer $b \in [0..maxgap(\phi)]$ or $x_i + maxgap(\phi) < x_{i+1}$, and (iv) $p \wedge \phi$ is satisfiable. Define $Prim(\phi)$ as the set of all primitive constraints for ϕ with respect to some enumeration of $var(\phi)$.

Because constraints are linear inequalities, applying the Fourier-Motzkin elimination method [29] to the gap atoms of p and ϕ yields $True$ if and only if $p \wedge \phi$ is satisfiable, i.e., condition (iv).

Example 3.12 : Consider the constraint

$$\phi_{\text{START}} = \{\texttt{Request}@x, \texttt{Schedule}@y, \texttt{Compute}@z, x+2 \leqslant z, y+4 \geqslant x, y+7 \geqslant z\}$$

The constraint $p = \{\texttt{Request}@x, \texttt{Schedule}@y, \texttt{Compute}@z, x+8 < y, y+1 = z\}$ is primitive for ϕ_{START} with respect to the enumeration x, y, z. ▮

Lemma 3.6 : Let ϕ be a constraint. The following hold:

1. $|Prim(\phi)|$ is $O(|\phi|! \cdot 2^{|\phi|^2})$, i.e., $Prim(\phi)$ is finite, and

2. $\phi \equiv \bigvee Prim(\phi)$.

where $\bigvee Prim(\phi)$ is the disjunction of all elements in $Prim(\phi)$.

Proof: (1) follows from definition of primitive constraints; there are $|var(\phi)|!$ enumerations of $var(\phi)$. For each gap between the $|var(\phi)| - 1$ pairs of consecutive variables in an enumeration, there are $maxgap(\phi) + 2$ possible gaps. Then, there are, at most, $(|maxgap(\phi)| + 2)^{(|var(\phi)|-1)}$ primitive constraints for each of the $|var(\phi)|!$ enumerations. Note that $var(\phi) \leqslant |\phi|$ and $maxgap(\phi) \leqslant 2^{|\phi|}$, so $|Prim(\phi)| \leqslant |\phi|! \cdot (2^{|\phi|} + 2)^{|\phi|-1}$. Then, $|Prim(\phi)|$ is $O(|\phi|! \cdot 2^{|\phi|^2})$.

For (2), assume for some assignment σ, $\eta \models \phi[\sigma]$. Without loss of generality, there is an enumeration e of $\mathit{var}(\phi)$ and a set of gap atoms bounded by $\mathit{maxgap}(\phi) + 1$ between consecutive variables in e such that σ satisfies these gap atoms. This enumeration and set of gaps, along with ϕ's event atoms, form a primitive constraint p in $\mathit{Prim}(\phi)$ such that $\eta \models p[\sigma]$.

Alternatively, assume $\bigvee \mathit{Prim}(\phi)$ is satisfied by some assignment σ. Then, there is some primitive constraint p satisfied by σ. Let g be an arbitrary gap atom in ϕ with variables x and y. Note that p fixes the gap between x and y as either exactly an integer from $[0..\mathit{maxgap}(\phi)]$ or at least $\mathit{maxgap}(\phi)+1$. By definition, $\phi \wedge p$ is satisfiable, so this gap is consistent with g. Thus, σ satisfies g. Finally, ϕ and p contain the same set of event atoms. Then, $\eta \models \phi[\sigma]$. ▮

A key observation in the proof of Lemma 3.4 is the assignment identifies timestamps where subformulas of AcycConn are satisfied, which correspond to timestamps for event atoms. A second important observation is that if each (non-root) node in T^x_ϕ must take a timestamp greater than or equal to its parent for ϕ to be satisfied, then AcycConn(T^x_ϕ, x) has no past-time LTL operators, which followed from the definition of the Gap function. The second observation is used later in a translation without past operators for a subclass of rules (Theorem 3.12).

Note that Lemma 3.6 allows decomposing all constraints into a finite set of primitive constraints. Each primitive constraint is acyclic and connected. Then, translating an arbitrary constraint can use AcycConn applied to each primitive constraint, joining the resulting LTL formulas with disjunction.

Let ϕ be a constraint and x an arbitrary variable in $\mathit{var}(\phi)$ with T^x_ϕ a derived tree. The following definition for a function ConsToLTL maps ϕ and x to an LTL formula:

$$\mathrm{ConsToLTL}(\phi, x) = \bigvee\nolimits_{p \in \mathit{Prim}(\phi)} \mathrm{AcycConn}(T^x_p, x)$$

The following property of ConsToLTL characterizes one way it can be applied.

Lemma 3.7 : Let η be an enactment, ϕ a constraint, x a variable in $\mathit{var}(\phi)$, and $i \in \mathbb{N}$ a timestamp. The following are equivalent:

1. There is some assignment σ such that $\eta \models \phi[\sigma]$ and $\sigma(x) = i$.

2. $\eta, i \models \mathrm{ConsToLTL}(\phi, x)$

Proof: By Lemma 3.6, a constraint ϕ is equivalent to a disjunction of its primitive constraints. Using an arbitrary variable x in ϕ to root the AcycConn translation, Lemma 3.4 equates the satisfaction of a primitive constraint p containing variable x with the satisfaction of the LTL formula $\mathrm{AcycConn}(T_p^x, x)$. Finally, $\mathrm{ConsToLTL}(\phi, x)$ collects the AcycConn translations of the primitive constraints for ϕ as a disjunction. ▌

ConsToLTL translates rules that share no variables in the following way: let $\phi \rightarrow \psi$ be a singly-linked rule whose constraints share no variables. Let y, z be arbitrary variables in $\mathit{var}(\phi), \mathit{var}(\psi)$, respectively. According to the rule semantics, when ϕ is satisfied by some assignment σ_ϕ with $\sigma_\phi(y) = i$ for some $i \in \mathbb{N}$, there must be an assignment σ_ψ that satisfies ψ such that $\sigma_\psi(z) = j$ for some $j \in \mathbb{N}$. By Lemma 3.7, this corresponds to the condition that when $\mathrm{ConsToLTL}(\phi, y)$ is satisfied at instant i, $\mathrm{ConsToLTL}(\psi, z)$ is satisfied at instant j. Because y is arbitrary variable in ϕ and is not used in ψ, the instant j satisfying $\mathrm{ConsToLTL}(\psi, z)$ is not necessarily related to i. Then, enforcing $\phi \rightarrow \psi$ is equivalent to checking that if $\mathrm{ConsToLTL}(\phi, y)$ is satisfied (somewhere), then $\mathrm{ConsToLTL}(\psi, z)$ is also satisfied (somewhere). It follows that:

Lemma 3.8 : Let η be an enactment, $\phi \rightarrow \psi$ a singly-linked rule with no shared variables, y a variable in $\mathit{var}(\phi)$, and z a variable in $\mathit{var}(\psi)$. The following are equivalent:

1. $\eta \models \phi \rightarrow \psi$

2. $\eta, 0 \models \mathsf{F}\,\mathrm{ConsToLTL}(\phi, y) \rightarrow \mathsf{F}\,\mathrm{ConsToLTL}(\psi, z)$

Proof: Let σ be an assignment. Assume $\eta \models \phi[\sigma]$. By Lemma 3.7 the trace η satisfies $\text{ConsToLTL}(\phi, y)$ for some instant $i \in \mathbb{N}$, i.e. $\eta, 0 \models \mathsf{F}\text{ConsToLTL}(\phi, y)$. If $\eta \models \phi \rightarrow \psi$, there must be an assignment σ' such that $\sigma'(y) = j$ for some timestamp $j \in \mathbb{N}$. Equivalently, by Lemma 3.7, the trace η must satisfy $\text{ConsToLTL}(\psi, z)$ at instant j, i.e. $\eta, 0 \models \mathsf{F}\text{ConsToLTL}(\psi, z)$.

The converse follows from the same reasoning. ▮

The above translation applies to rules that share no variables. Next we translate singly-linked rules whose constraints share one variable.

ConsToLTL translates rules whose constraints share one variable as follows: let $\phi \rightarrow \psi$ be a singly-linked rule whose constraints share the variable y. By Lemma 3.7, if $\text{ConsToLTL}(\phi, y)$ and $\text{ConsToLTL}(\psi, y)$ are satisfied at the same timestamp i in a trace, there is a pair of assignments σ_ϕ and σ_ψ that satisfy ϕ and ψ respectively, such that these assignments agree on y, i.e. $\sigma_\phi(y) = \sigma_\psi(y) = i$. Then, to identify satisfying assignments for ϕ and ψ that agree on y, the formulas $\text{ConsToLTL}(\phi, y)$ and $\text{ConsToLTL}(\psi, y)$ are placed in the same temporal scope.

This motivates translating a singly-linked rule $\phi \rightarrow \psi$ with common variable y to an LTL formula $\text{ConsToLTL}(\phi, y) \rightarrow \text{ConsToLTL}(\psi, y)$. The global operator G ensures the LTL formula covers all timestamps y can take.

Theorem 3.9 : Let η be an enactment and $\phi \rightarrow \psi$ a singly-linked rule with a shared variable y. The following are equivalent:

1. $\eta \models \phi \rightarrow \psi$
2. $\eta, 0 \models \mathsf{G}(\text{ConsToLTL}(\phi, y) \rightarrow \text{ConsToLTL}(\psi, y))$

Proof: Let $i \in \mathbb{N}$ be a timestamp and σ be an assignment such that $\sigma(y) = i$. Assuming $\eta \models \phi[\sigma]$, by Lemma 3.7 the trace η satisfies $\text{ConsToLTL}(\phi, y)$ at instant i. If $\eta \models \phi \rightarrow \psi$, there must be an assignment σ' such that $\sigma'(y) = i$ and $\eta \models \psi[\sigma']$. According to Lemma 3.7, the trace η must satisfy $\text{ConsToLTL}(\psi, y)$ at instant i, leading to $\eta, i \models \text{ConsToLTL}(\phi, y) \rightarrow \text{ConsToLTL}(\psi, y)$. The enclosing G ensures that the above holds for all such timestamps i. ∎

In the remainder of the section, we develop a translation SingToLTL for the entire subclass of singly-linked rules. We start by looking at a singly-linked rule with a primitive body. Let $p \rightarrow \psi$ be a singly-linked rule where p is a primitive constraint. Without loss of generality, let $x_1, ..., x_n$ be the enumeration in p. For each subset $V \subseteq \mathit{var}(\phi)$, $\phi|_V$ denotes the constraint obtained from ϕ after removing all atoms involving a variable not in V. Note that $p|_{x_1,...,x_n} = p$. For each $j \in [1..n]$, define the following function SingToLTL to map $p|_{x_j,...,x_n} \rightarrow \psi$ to an LTL formula $\text{SingToLTL}(p|_{x_j,...,x_n} \rightarrow \psi)$:

- $\text{AcycConn}(T_p^{x_j}, x_j) \rightarrow \text{ConsToLTL}(\psi, x_j)$ if the shared variable is x_j
- $(\bigwedge_{r@x_j \in p} r) \rightarrow \mathsf{X}^b \, \text{SingToLTL}(p|_{x_{j+1},...,x_n} \rightarrow \psi)$ if the shared variable is not x_j and p contains $x_j + b = x_{j+1}$
- $(\bigwedge_{r@x_j \in p} r) \rightarrow \mathsf{X}^{b+1} \mathsf{G} \, \text{SingToLTL}(p|_{x_{j+1},...,x_n} \rightarrow \psi)$ if the shared variable is not x_j and p contains $x_j + b < x_{j+1}$

Note that SingToLTL only introduces past-time operators X^{-1} and P within AcycConn and ConsToLTL. The SingAcycToLTL translation uses both the future-time LTL operators, X and F and the past-time operators X^{-1} and P. Many rules take the form of "if a condition is observed, then another condition must be observed later"; we call these *future rules*, also known as *strictly sequential rules* [30].

Definition : A singly-linked, dataless rule $\phi \rightarrow \psi$ whose constraints share exactly one variable y is a *future rule* if ψ implies $y \leqslant z$ for all $z \in \mathit{var}(\psi)$.

Such rules only reference events matching the rule head whose timestamps are greater than those for the body. This new function SingToLTL avoids the past-time operators X^{-1} and P for "future" singly-linked rules. This observation is crucial in establishing Theorem 3.11. We prove that SingToLTL produces a correct translation in Lemma 3.10.

We use the following refined notion of satisfaction: Let σ be an assignment. An enactment η *satisfies* $\phi \rightarrow \psi$ *for* σ if $\eta \models \phi[\sigma]$ implies the existence of an assignment σ' such that $\eta \models \psi[\sigma']$ and both σ, σ' agree on $var(\phi)$.

Lemma 3.10 : Let η be an enactment, $\phi \rightarrow \psi$ a singly-linked rule with ϕ, ψ sharing the variable x_k ($k \in [1..n]$), p a primitive constraint for ϕ with respect to the enumeration $x_1, ..., x_n$, j an integer with $j \in [1..k]$, and i a timestamp in η. The following are equivalent:

1. $\eta \models p|_{x_j,...,x_n} \rightarrow \psi$ for each assignment that assigns i to x_j
2. $\eta, i \models \text{SingToLTL}(p|_{x_j,...,x_n} \rightarrow \psi)$

Proof: The proof is performed by induction on j with the base case of $j = k$. The base case corresponds to the first case in the definition for SingToLTL and follows from similar reasoning as Theorem 3.9. The inductive steps holds by the following reasoning: Let $p|_{x_j,...,x_n}$ be a constraint $\{r_j@x_j, x_j{+}b = x_{j+1}, ...\}$. Assume the event atom $r_j@x_j$ is satisfied by η and an assignment that maps x_j to i, checking $p|_{x_j,...,x_n} \rightarrow \psi$ for assignments that extend $[x_j \mapsto i]$ reduces to checking $p|_{x_{j+1},...,x_n} \rightarrow \psi$ for assignments that map x_{j+1} to $i{+}b$, because the event atom $r_j@x_j$ and equality $x_j{+}b = x_{j+1}$ complete the initial portion $\{r_j@x_j, x_j{+}b = x_{j+1}\}$ in $p|_{x_j,...,x_n}$. The operators X^b prefix the translation of $p|_{x_{j+1},...,x_n} \rightarrow \psi$ at instant $i{+}b$. Similarly, when p contains $x_j{+}b < x_{j+1}$ and the event atom $r_j@x_j$ is satisfied by η with an assignment that maps x_j to i, checking $p|_{x_j,...,x_n} \rightarrow \psi$ for assignments that extend $[x_j \mapsto i]$ reduces to checking $p|_{x_{j+1},...,x_n} \rightarrow \psi$ for assignments that map x_{j+1} to some timestamp $>i{+}b$. The operators $\mathsf{X}^{b+1}\mathsf{G}$ to prefix the translation of $|p_{x_{j+1},...,x_n} \rightarrow \psi$ at all instants later than $i{+}b$. ▌

We now present the following theorem stating the equivalence of a singly-linked, dataless rule and its translation using primitive constraints in Theorem 3.9.

Theorem 3.11 : Let η be an enactment and $\phi \rightarrow \psi$ a singly-linked, dataless rule. The following are equivalent:

1. $\eta \models \phi \rightarrow \psi$

2. $\eta, 0 \models \bigwedge_{p \in Prim(\phi)} \mathsf{G}\, \text{SingToLTL}(p \rightarrow \psi)$

and $|\bigwedge_{p \in Prim(\phi)} \mathsf{G}\, \text{SingToLTL}(p \rightarrow \psi)|$ is $O(|\phi|! \cdot 2^{|\phi|^3} \cdot |\phi| + |\phi|! \cdot |\psi|! \cdot 2^{|\psi|^4 + |\phi|^2})$.

Proof: The constraint ϕ is equivalent to the disjunction of $Prim(\phi)$ by Lemma 3.6; thus checking $\phi \rightarrow \psi$ can be accomplished by checking $p \rightarrow \phi$ for each $p \in Prim(\phi)$. By Lemma 3.10, SingToLTL$(p \rightarrow \psi)$ holds at instant i if and only if $p \rightarrow \psi$ is satisfied for each assignment that assigns timestamp i to the first variable in p. Finally, $p \rightarrow \psi$ is enforced for all assignments, by enforcing SingToLTL$(p \rightarrow \psi)$ at all instants in η with G. ▮

Now we show that this translation of future rules does not contain any past LTL operators.

Theorem 3.12 : Let $\phi \rightarrow \psi$ be a future, singly-linked rule. Then, the LTL formula $\bigwedge_{p \in Prim(\phi)} \mathsf{G}\, \text{SingToLTL}(p \rightarrow \psi)$ contains no X^{-1} nor P operators.

Proof: Consider the definition of SingToLTL function and let x_k be the shared variable in $\phi \rightarrow \psi$. First, for a primitive constraint p of ϕ, the translation function SingToLTL$(p \rightarrow \psi)$ unrolls p until it reaches x_k, inserting only X and G, i.e. only future LTL operators.

Second, SingToLTL translates the portion of p starting at x_k as AcycConn$(p|_{x_k,\ldots,x_n}, x_k)$, where x_k is the smallest variable in $p|_{x_k,\ldots,x_n}$. Similarly, the head of the rule is translated as ConsToLTL(ψ, x_k), which applies AcycConn to each element of $Prim(\psi)$ and x_k. The property of being a future rule implies that x_k is the smallest variable in ψ, i.e., x_k is the smallest variable in each constraint in $Prim(\psi)$. By virtue of its construction, AcycConn inserts only X and F when translating a primitive constraint from its smallest variable. Thus, SingToLTL$(p \rightarrow \psi)$ has no X^{-1} nor P, i.e. no past-time, operators. ▮

Size of Translation: Finally, we establish the size of $\bigwedge_{p \in Prim(\phi)} \mathsf{G}\, \text{SingToLTL}(p \rightarrow \psi)$. For each primitive constraint p of ϕ, SingToLTL unfolds p: p has $|var(\phi)|$ variables, so unfolding yields at most $(|\mathit{maxgap}(\phi)| + 1) \cdot (|\mathit{var}(\phi)| - 1)$ next operators, at most $|\mathit{var}(\phi)| - 1$ global operators, propositions, conjunction, and implication operators each. SingToLTL also inserts ConsToLTL(ψ) at the variable shared with p: applying $|\mathit{Prim}(\psi)|$ from Lemma 3.6 with the size of AcycConn established in the proof of Theorem 3.5, it follows that $|\text{ConsToLTL}(\psi)|$ is $O(|\psi|! \cdot (2^{|\psi|})^4)$. Note that $\mathit{var}(\phi) \leqslant |\phi|$ and $\mathit{maxgap}(\phi) \leqslant 2^{|\phi|}$, and similarly for ψ. Then, $|\text{SingToLTL}(p \to \psi)|$ is $O(2^{|\phi|} \cdot |\phi| + |\psi|! \cdot 2^{|\psi|^4})$. Applying Lemma 3.6 capture the conjunction of all p in $\mathit{Prim}(\phi)$, the SingToLTL translation of $\phi \to \psi$ is $O(|\phi|! \cdot 2^{|\phi|^3} \cdot |\phi| + |\phi|! \cdot |\psi|! \cdot 2^{|\psi|^4+|\phi|^2})$. In summary, this translation for singly-linked, dataless rules is double-exponential in the size of the rule.

This concludes our the translation of singly-linked, dataless rules. Next, we compare the results of this translation and that of the previous section with a translation based on Kamp's Theorem, showing that both translations lower the complexity of the size of the resulting LTL formula.

3.5 Related Work

The technical problem concerning translation of (super)classes of rules into LTL is not new and has been discussed variously [31–33], and in particular by Kamp [25], who shows: *Given any first-order monadic logic of order formula with one free variable, there is a temporal logic [LTL] formula which is equivalent over Dedekind-complete chains.* The natural numbers are a Dedekind-complete chain and the free variable is interpreted as the first timestamp in a trace, so this result indicates the expressive equivalence of first-order logic with monadic predicates and ordering $<$ (FOMLO) and LTL with past-time operators over the natural numbers. Dataless rules uses event types with no attributes, thus these event types constitute monadic predicates over the the natural numbers, and gap atoms can be expressed with ordering relations. Additionally, each rule has the form $\forall \bar{x}.\bar{y}.(\phi(\bar{x}, \bar{y}) \to \exists \bar{z}.\psi(\bar{y}, \bar{z}))$, and a set of rules is interpreted as a conjunction, thus dataless rules use the same vocabulary as FOMLO. Thus, the following holds by Kamp's Theorem:

Theorem 3.13 : [25] Every set of dataless rules has an equivalent LTL formula.

Rabinovich [26] presents a concise proof of Kamp's Theorem which provides key pieces of an FOMLO-to-LTL translation algorithm. The algorithm applies recursively to FOMLO formulas, with the help of a set of auxiliary formulas that impose total orderings on the variables in the FOMLO formula. This technique is similar to our translation in Section 3.4, which imposes total orderings on variables. However, in the translation algorithm in [26], FOMLO negation triggers an exponential increase in the size of the result for each quantifier alternation. This yields LTL formulas with size hyper-exponential (greater than any exponential function) in the size of the input FOMLO formula. In comparison, our translations produce equivalent LTL formulas that are single- and double-exponential in the size of the input. Thus, our translations improve the size complexity of the resulting LTL formulas, which is important for the size of the resulting automata for runtime monitoring.

Section 3.1 describes the use of finite state machines to detect violations. Many techniques already exist for monitoring event streams against formal specifications, especially properties like relative ordering, e.g., those easily expressible by LTL or automata. DECLARE [34] is the most common formalism for expressing business rules in the business process management community and has semantics grounded in future-time LTL. Reference [35] exhibits runtime monitors for DECLARE constraints with a translation through LTL to finite state automata. For general software systems, [36] tracks assignments to subformula of properties for monitoring first-order past LTL properties. A limitation of these works is that past- and future-time LTL, DECLARE, and Linear Dynamic Logic (LDL) lack an immediate representation of quantitative time constraints, e.g., `A` *must occur within 5 days of* `B` *and* `C` *must occur within 2 days of* `B` *and within 3 days of* `A`, which must be carefully encoded in automata as the number of quantitative relationships grows. More study is needed to understand which quantitative time constraints can be expressed in these existing non-quantitative languages; this chapter is a step in that direction. Alternatively, LTL and LDL have disjunction and negation operators; it is not clear what expressivity these operators would add to rules.

DECLARE is extended with quantitative time constraints between pairs of activities [37,38]. Reference [37] provides monitoring algorithms derived from Abductive Logic Programming. The size of the auxiliary storage used by the resulting monitors is unbounded with respect to the size of the execution trace; *trace-length independence*, where memory usage is bounded regardless of the target trace, is a desirable property of runtime monitors [17]. In our approach, the size of the monitor (a finite state machine) for a given set of rules is fixed. References [38] and [39] monitor timed DECLARE constraints by translating them to Metric Temporal Logic, which has a translation to timed automata, and event calculus, respectively. Our translation from rules to untimed LTL allows for monitoring by deterministic finite automata, a simpler model of computation than timed automata. Additionally, DECLARE constraints are built from binary relationships between activities, while rules can express pre- and post-conditions with an arbitrary number of event and gap atoms.

Some temporal logics with quantitative operators have been studied as specification languages for monitoring general software systems. Reference [40] employs formula rewriting to track satisfaction for Mission-time Linear Temporal Logic. Reference [41] employs formula rewriting to monitor the subclass of Metric Temporal Logic with bounded future formulas, i.e., all temporal modalities have a deadline, which is not a restriction of our rules. Reference [42] constructs deterministic automata for the past-time subset of Metric Temporal Logic where all temporal modalities have upper and lower natural number bounds. Rules can reference the indefinite past and future of an enactment with respect to each timestamp, so they do not have these restrictions. More generally, it is not clear which quantitative time features are best suited for specifying business rules.

There are many more approaches to enforcing constraints on streams of events that are not based on runtime monitoring. Simple Temporal Networks (STNs) [43] use networks of events where edges represent relative time constraints. STNs can be extended to Conditional Simple Temporal Networks with Uncertainty (CSTNUs) with the conditional constraints using some propositional variables and uncertainty as ranges for timestamps. References [44–46], and study workflows subject to constraint networks for controllability: determining if the workflow can be enacted to satisfy the network regardless of the duration of the uncontrollable gaps between events. [47] checks controllability for the the Guard Stage Milestone language extended with time constraints. The controllability problem is different from our problem: controllability decides the potential for constraint satisfaction at design-time, while our approach decides the observed satisfaction of constraints at runtime. Additionally, static verification techniques and controllability both assume a process model or models for the service exist; this chapter assumes no process model, only a set of event types being completed, which affords more flexibility in choosing which events to monitor.

3.6 Chapter Summary

This chapter studies the early violation detection problem for dataless rules. We describe an approach based on translating quantitative constraints to finite state machines. We present two translations from rules to LTL formulas, one for singly-linked, acyclic, dataless rules with the correctness and single-exponential size of the translation established, and one for singly-linked, dataless rules with the correctness and double-exponential size of the translation established.

Several related problems deserve more research. First, exploring the semantics of time in formal specification languages as well as industry standards may improve constraint specification. Allowing explicit timestamps (e.g., rules with timestamps) or implicit timestamps (e.g., LTL), past constraints or future constraints, and discrete or continuous time, lead to different suitabilities for applications and may incur different complexity for violation detection. Also, it may be possible to translate dataless rules directly to automata of smaller size or to demonstrate that certain sizes of automata are optimal, while translating larger classes of dataless rules may enable violation detection wider classes of applications. Finally, referencing data in rules is critical for matching information between events; the subsequent chapters of this dissertation study rules with data.

Chapter 4

Rules with Data

In this chapter, we address the problem of detecting violations for rules with data. Adding data variables over a data domain expands the types of properties rules can express, such as matching the user of a `Payment` event with the user who made the `Request`. First, we consider the problem for an individual rule, developing a technique for calculating the earliest time a violation is inevitable (the "deadline") and use this time for detecting violations of individual rules. Then, we observe that interactions within a set of rules creates situations where an enactment may violate a set of rules though no individual rule is violated, which is trivial in the absence of data but non-trivial with data, so we extend our algorithms to handle a set of rules by simulating the effects of rule interaction using a chase process. To ensure chase termination, the chapter's results for multi-rule violation detection are limited to "acyclic" sets of rules. We also present two optimizations to reduce computational overhead of the violation detection algorithms. Finally, we evaluate the feasibility of our techniques to determine where our approach is beneficial; we implement the individual rule and multi-rule algorithms and characterize their performance on a variety of enactments and rule sets.

This chapter is organized as follows. Section 4.1 presents the Deadline algorithm for computing deadlines, then the data structures and Update, Update-E, Build, and Detect algorithms for storing relevant event data and detecting violations. Section 4.2 extends these techniques to acyclic sets of rules with the Chase and Detect-Multi algorithms. Section 4.3 presents two optimization techniques for Update and Update-E. Section 4.4 provides key findings of evaluations of an implementation of the algorithms. Finally, Sections 4.5 and 4.6 discuss related work and conclude the chapter.

4.1 Algorithms for Individual Rules

In this section, we develop key techniques for early violation detection for individual rules. First, we define the concept of deadlines and present an algorithm to calculate deadlines. We then define data structures to store variable assignments and present algorithms to create new assignments from arriving events, and to merge and match existing assignments. Finally, we detail how violations are detected using these algorithms.

We aim to detect violations at the earliest possible time. Since an enactment is an accumulation of events with increasing timestamps, it may be that a complete body assignment derived from the current enactment can only be extended at or before a specific future time called a deadline. We now formulate the notion of a deadline below.

Definition : Let Θ be a set of gap atoms over variables $x_1, ..., x_n$ and μ an assignment for variables x_i's. We use $\textsc{def}_\mu$ for the variables μ assigns a value; $\mu(\Theta)$ the gap atoms obtained by replacing each variable $x \in \textsc{def}_\mu$ with $\mu(x)$, and $\max(\mu) = \max\{\mu(x) \mid x \in \textsc{def}_\mu\}$. A timestamp $\tau \in \mathbb{N}$ is the *deadline* for $\Theta, x_1, ..., x_n, \mu$ if (1) $\tau \geqslant \max(\mu)$, and (2) either $\mu(\Theta)$ is unsatisfiable and $\tau = \max(\mu)$ or conditions (i) and (ii) both hold: (i) for each complete extension μ' of μ such that $\mu'(x) > \tau$ for each $x \notin \textsc{def}_\mu$, $\mu'(\Theta)$ is *false*, and (ii) there is a complete extension μ'' of μ such that $\mu''(\Theta)$ is *true*.

Example 4.1 : For the example enactment introduced in Section 2.3 and rule r_1 in Fig. 4.2, the body assignment μ_{10} is created at time 8, where $\mu_{10}(x)$=3, $\mu_{10}(y)$=6, and

Request

ID	*user*	*account*	ts
π_1	Alice	a3	1
π_1	Alice	a4	3
π_2	Bob	b6	7

Approval

ID	*user*	ts
π_1	Alice	6

Reserve

ID	*user*	*account*	ts
π_1	Alice	a4	8
π_1	Alice	a3	9

Payment

ID	*user*	*account*	ts
π_1	Alice	a3	8
π_1	Alice	a4	9

Launch

ID	*user*	*account*	ts

Figure 4.1: Database S_9 with events from two enactments π_1 and π_2.

$$r_1 : \ \texttt{Request}(u,a)@x, \texttt{Approval}(u)@y, x \leqslant y \leqslant x+7, \texttt{Reserve}(u,a)@z, y \leqslant z \leqslant \ y+7$$

$$\rightarrow \texttt{Payment}(u,a)@w, \texttt{Launch}(u,a)@v, y \leqslant w \leqslant y+3, z \leqslant v \leqslant z+7, v \leqslant w+4$$

Figure 4.2: Rule r_1

$\mu_{10}(z)$=8. As shown in Fig. 2.2, applying μ_{10} to the head atoms yields upper bounds $w \leqslant 9$ (=y+3) and $v \leqslant 15$ (=z+7). These bounds show that extensions of μ_{10} must have a `Payment` event whose time variable w is no later than time 9. Thus, the time 9 is a *deadline* for μ_{10}: the latest time μ_{10} can be extended w.r.t. w, and the earliest time μ_{10} could be recognized as a witness of a violation. Fortunately, a `Payment` event happened at time 9, which satisfies $w \leqslant 9$. However, v remains unresolved and thus the subsequent deadline to extend μ_{10} is the latest time to observe a value for v: $v \leqslant 13$ (=w+4) and $v \leqslant 15$ (=z+7), so the deadline to extend μ_{10} is changed to 13. ▮

We compute deadlines with function Deadline (Alg. 1). Deadline determines for each x_i the least τ_i such that $\mu(\Theta) \rightarrow x_i \leqslant \tau_i$, and the deadline τ is the least of τ_i's. First, if $\mu(\Theta)$ is unsatisfiable, μ is a violation at the time of its creation, i.e., at its largest timestamp. Otherwise, an array *UpperBd* is initialized with constants (Lines 3-5), then tightened with the initial bounds and the gap atoms in Θ: a gap atom $u \pm k \leqslant v$ indicates *UpperBd*$(v) \mp k$ is an upper bound for u. For each gap atom $u \pm k \leqslant v$ for which *UpperBd*(v) is defined, we update *UpperBd*(u) as max(*UpperBd*$(v) \mp k$, *UpperBd*(u)) (Lines 7-9).

The Deadline function (Alg. 1) can compute deadlines for complete body assignments and for complete body assignments with matching partial head assignments. For a complete body assignment μ and a partial head assignment β, we compute the latest time $\mu \cup \beta$ can be extended. This time is, in fact, the earliest time μ becomes a witness for a violation. In the following lemma, we state a property of deadlines for a complete body

Algorithm 1 Deadline($\Theta, x_1, ..., x_n, \mu$)

```
Input: A set of gap atoms $\Theta$ over time variables $x_1, ..., x_n$ and an assignment $\mu$
Output: A timestamp $\tau$
if If $\mu(\Theta)$ is unsatisfiable then return $\tau := \max(\mu)$;
                                  /* $\max(\mu)$ is the largest timestamp $\mu$ assigns to $x_1, ..., x_n$*/
end if
Rewrite each atom in $\mu(\Theta)$ in the form $u \pm k \leqslant v$;
                                  /* $u, v$ either a time variable or in $\mathbb{N}$, $k \in \mathbb{Z}$ */
Let $UpperBd$ be a map from $x_1, ..., x_n$ to $\{\infty\}$;
for each $u \pm k \leqslant v$ in $\mu(\Theta)$ with $v \in \mathbb{N}$ and $u \in \{x_1, ..., x_n\}$ do
    $UpperBd(u) := v \mp k$ ;
end for
for $|\Theta|$ iterations do
    for each gap atom $u \pm k \leqslant v$ in $\mu(\Theta)$ do
        if $UpperBd(v)$ is finite and $UpperBd(u) \pm k > UpperBd(v) \geqslant 0$ then
            $UpperBd(u) := UpperBd(v) \mp k$ ;
        end if
    end for
end for
return $\tau := \min\{UpperBd(x_i) \mid 1 \leqslant i \leqslant n\}$
```

assignment and partial head assignment.

Lemma 4.1 : Let r: $\varphi \rightarrow \psi$ be a rule, φ_g, ψ_g the gap atoms in φ, ψ (resp.), μ a (body) assignment such that $\mu(\varphi_g)$ is *true*, β an incomplete head assignment matching μ such that $\beta(\mu(\psi_g))$ is satisfiable, and U the variables in ψ_g undefined by β. Let $\tau = \mathsf{Deadline}(\psi_g, \mathit{var}(\varphi_g \cup \psi_g), \mu \cup \beta)$. The following hold:

1. If $\tau \in \mathbb{N}$, then there is a head assignment β' matching $\mu \cup \beta$ such that $\min(\beta'(U)) \leqslant \tau$ and $\beta'(\psi_g)$ is *true*,
2. If $\tau \in \mathbb{N}$, then for all complete head assignments β' matching $\mu \cup \beta$ such that $\max(\beta'(U)) > \tau$, $\beta'(\psi_g)$ is *false*, and
3. If $\tau = \infty$, then for all timestamps n in $\mathbb{N}$, there is a head assignment β' matching $\mu \cup \beta$ such that $\max(\beta'(U)) > n$ and $\beta'(\psi_g)$ is *true*.

Proof: To show (1), assume there is no complete head assignment β' extending $\mu \cup \beta$ such that $\min(\beta'(U)) \leqslant \tau$ and $\beta'(\psi_g)$ is *true*. Then, $(\mu \cup \beta)(\psi) \wedge (z = \tau)$ is not satisfiable. Then, there is a gap atom in $\mu \cup \beta(\psi)$ that provides an upper bound for z below τ. Then, τ is not the minimum of the upper bounds in *UpperBd*. Thus Algorithm 1 on $\mu \cup \beta$ and ψ should not output τ. This is a contradiction. To show (2), assume some complete head assignment β' extends $\mu \cup \beta$ such that $\min(\beta'(U)) > \tau$ and $\beta'(\psi_g)$ is true. Then, $(\mu \cup \beta)(\psi) \wedge (z = \tau')$ is satisfiable for some z in $var(\psi)$. Then, $\mu(\psi)$ does not imply $z_i \leqslant \tau$ for all variables z_i. Thus Algorithm 1 on $\mu \cup \beta$ and ψ should not output τ. This is a contradiction. To show (3), assume $\tau = \infty$. Algorithm 1 only produces ∞ when $\mu(\psi)$ is satisfiable and for some variable z_i and for all $n \in \mathbb{N}$, $\mu(\psi) \not\rightarrow (z_i \leqslant n)$ Then, for all timestamps n in $\mathbb{N}$, there is some complete assignment that extends μ, satisfies ψ, and uses some n' larger than n. Finally, μ can be extended arbitrary far in the future. ▮

The discussions in Section 2.3 suggest maintaining partial and complete assignments for rule variables. We define three tabular data structures: BA_r for body assignments, HA_r for head assignments, and EXT_r (extensions), to track pairings of body and head assignments. BA_r and HA_r consist of the following columns: (*i*) one column for the assignment identifier (*Aid*) from **I**, (*ii*) one column for the enactment identifier (ID) from **I**, (*iii*) one column in BA_r for each variable in φ and one column in HA_r for each event variable in ψ (resp.) (a variable in the head ψ is an *event variable* if it occurs in an event atom in ψ.) to hold a value from $\mathcal{D}$ or a timestamp, and (*iv*) one column for gap atoms in φ and ψ (resp.) simplified with the assigned values as possible. Additionally, BA_r has one more column (*v*) *match?* indicating with *yes* or *no* the presence or absence, resp., of a complete head assignment matching the complete body assignment. For convenience, we refer to rows in these two tables as assignments. EXT_r has three columns: (*i*) one column for a body *Aid* from BA_r, (*ii*) one column for a head *Aid* from HA_r that extends the row's body assignment, and (*iii*) one column for the *deadline*, calculated using the row's assignments and the head gap atoms as inputs for Deadline.

For each enactment η, $\text{BA}_r(\eta)$ and $\text{HA}_r(\eta)$ store all assignments that can be generated from set of events in η and satisfy φ and ψ (resp.). Specifically, for a rule $r{:}\varphi \rightarrow \psi$ and an enactment η, $\text{BA}_r(\eta)$ contains every assignment μ such that for a non-empty subset P of the event atoms in φ, μ is defined for the variables in P, $\mu(P) \subseteq \eta$, and for each gap atom

Aid	ID	u	a	x	y	z	
μ_1	π_1	Alice	a3	1	-	-	
μ_2	π_1	Alice	a4	3	-	-	
μ_3	π_1	Alice	-	-	6	-	
μ_4	π_1	Alice	a3	1	6	-	$\mu_1 + \mu_3$
μ_5	π_1	Alice	a4	3	6	-	$\mu_2 + \mu_3$
μ_6	π_2	Bob	b6	7	-	-	
μ_7	π_1	Alice	a4	-	-	8	
μ_8	π_1	Alice	a4	3	-	8	$\mu_2 + \mu_7$
μ_9	π_1	Alice	a4	-	6	8	$\mu_3 + \mu_7$
μ_{10}	π_1	Alice	a4	3	6	8	$\mu_2 + \mu_9$
μ_{11}	π_1	Alice	a3	-	-	9	
μ_{12}	π_1	Alice	a3	1	-	9	$\mu_1 + \mu_{11}$
μ_{13}	π_1	Alice	a3	-	6	9	$\mu_3 + \mu_{11}$
μ_{14}	π_1	Alice	a3	1	6	9	$\mu_4 + \mu_{12}$

(a) All body assignments at time 9

Aid	ID	u	a	x	y	z	
μ_{15}	π_1	Alice	-	-	10	-	
μ_{16}	π_1	Alice	a4	3	10	-	$\mu_2 + \mu_{15}$
μ_{17}	π_2	Bob	-	-	10	-	
μ_{18}	π_2	Bob	b6	7	10	-	$\mu_6 + \mu_{17}$

(b) New body assignments added at time 10

Figure 4.3: Assignments for the rule body φ and events S_9, events $S_9 \cup \{e_1, e_2\}$

g whose variables appear in P, η satisfies g with μ. $\textsc{ha}_r(\eta)$ is similar, using ψ instead of φ. Fig. 4.4(a) shows the assignments inserted into $\textsc{ba}_r$ table at time 10 (those from Fig. 4.3(b)) with columns for gap atoms and the possibility of matching. $\textsc{ext}_r(\eta)$ stores each pair of assignments from $\textsc{ba}_r(\eta)$ and $\textsc{ha}_r(\eta)$, resp., such that the body assignment can be extended by the head assignment only at or before the row's deadline, i.e., some unknown variable in the body and head assignments is constrained by a gap atom in the body or head to be at or before the deadline.

Example 4.2 : Given the events up to time 9 in Fig. 4.1, suppose two events happen at time 10, e_1:`Approval`$(\pi_1, [\text{Alice}], 10)$ and e_2:`Approval`$(\pi_2, [\text{Bob}], 10)$. Event e_1 generates an assignment μ_{16}: $[\pi_1, \text{Alice}, \text{-}, \text{-}, 10, \text{-}]$, which combines with μ_2 into μ_{16}. Event e_2 yields new assignments μ_{17} and μ_{18}. Fig. 4.3(b) lists four assignments generated by e_1 and e_2. ▮

Aid	u	a	x	y	z	gap atoms	match?
μ_{10}	Alice	a4	3	6	8	-	No
μ_{11}	Alice	a3	-	-	9	$x \leqslant y \leqslant x+7$, $y \leqslant 9 \leqslant y+7$	No
μ_{12}	Alice	a3	1	-	9	$1 \leqslant y \leqslant 8$, $y \leqslant 9 \leqslant y+7$	No
μ_{13}	Alice	a3	-	6	9	$x \leqslant 6 \leqslant x+7$	No
μ_{14}	Alice	a3	1	6	9	-	No

(a) Some assignments in $\textsc{ba}_r(\pi_1)$ (Fig.4.3(a)) at $\mathsf{ts} = 9$

Aid	u	a	w	v	gap atoms
β_1	Alice	a3	8	-	$v \leqslant 12$
β_2	Alice	a4	9	-	$v \leqslant 13$
β_3	Alice	a3	-	12	$8 \leqslant w$
β_4	Alice	a3	8	10	-

(b) Some assignments in $\textsc{ha}_r(\pi_1)$ (Fig.4.3(b)) at $\mathsf{ts} = 10$

Figure 4.4: Body and Head Table Examples

We next present the `Update` algorithm to create and combine assignments with a batch of events, as shown in Example 4.2. This algorithm maintains $\textsc{ba}$ and $\textsc{ha}$ incre-

body Aid	head Aid	deadline
μ_{10}	-	9
μ_{10}	β_2	13
μ_{14}	-	9
μ_{14}	β_1	12
μ_{14}	β_3	12
μ_{14}	β_4	-

Figure 4.5: Extensions of μ_{10} and μ_{14} in $\text{EXT}_r(\pi_1)$ at $\mathsf{ts} = 13$

Algorithm 2 $\mathsf{Update}(\Theta, \Delta, T(\eta))$

```
Input: A set of atoms $\Theta$, a batch $\Delta$ for an enactment $\eta$, a table $T(\eta)$ ($T$ is $\text{BA}_r$ or $\text{HA}_r$)
Output: The updated table $T(\eta \cup \Delta)$ for $\eta \cup \Delta$
$\Gamma := T(\eta)$ ;
for each event $e \in \Delta$ do
    for each event atom $\gamma$ in $\Theta$ with the same name as $e$ do
        Create an assignment $\mu$ such that $\mu(\gamma) = e$ ;
        if $\mu(\Theta)$ is satisfiable then
            Add to $\Gamma$ the row $s = \langle a, e.\text{ID}, \mu(v_1), ..., \mu(v_n), \text{B}, (\text{no})\rangle$,
                where $a$ is a fresh assignment identifier, $v_1, ..., v_n$ are the event variables
                in $\Theta$, and B the gap atoms in $\Theta$, evaluated and simplified with $\mu$;
        end if
    end for
end for
while $\Gamma$ changes do
    for each pair of unique and consistent rows $\mu_1$ and $\mu_2$ in $T$ do
        $\mu$ :=merge$(\mu_1, \mu_2)$ ;                    /* consistent, merge explained in the text */
        if $\mu(\Theta)$ is satisfiable then
            Add to $\Gamma$ the row: $s = \langle a, \mu_1.\text{ID}, \max(t_1, t_2), \mu(v_1), ..., \mu(v_n), \text{B}, (\text{no})\rangle$
            where $a$ is a fresh assignment identifier and
            B is the union of gap atoms in $\mu_1, \mu_2$, evaluated with $\mu$ ;
        end if
    end for
end while
output $\Gamma$
```

mentally without accessing the corresponding enactment directly; this is important since enactments may be very large.

We now outline the behavior of Update. Given atoms Θ (here, the body or head of a rule), a batch Δ, and either BA_r or HA_r for an enactment η, First, the events in Δ and Θ are used to generate assignments, which are added to the table if they are consistent with Θ's gap atoms, (and thus extendible to complete assignments). Then, the **while** loop searches for pairs of consistent, partial assignments; two assignments are consistent if they agree on the variables for which they are both defined, e.g., in Fig. 4.3 μ_1 and μ_2 agree on u but not on a, so they are not consistent. For each pair of consistent assignments, a new assignment is created by combining their variable mappings and gap atoms and a deadline is computed. For example, assignment μ_5 in Fig. 4.3 is the *merge* of μ_2 and μ_3. If the new assignment is consistent with Θ, it is added to the table. The **while** loop considers only pairs of assignments pre-existing in Γ or Δ, i.e., it doesn't introduce new data values; this ensures that **while** loop terminates.

Example 4.3 : For the enactment and rule in Section 2.3, consider the enactment's event $\texttt{Request}(\pi_1, [\text{Alice}, \text{a3}], 1)$ and the rule's atom $\texttt{Request}(\text{user } u, \text{account } a)@x$. The mapping $[\textsc{id} \mapsto \pi_1, u \mapsto \text{Alice}, v \mapsto \text{a3}, x \mapsto 1]$ maps the atom to this event; the assignment corresponding to this mapping is added to $\textsc{ba}_r$ as μ_1 in Fig. 4.3(a). For the same example in Section 2.3 and Fig. 4.3(a), assignments μ_2: $[\pi_1, \text{Alice}, \text{a4}, 3, -, -, \{3 \leqslant y \leqslant 10, y \leqslant z \leqslant y+7\}]$ and μ_3: $[\pi_1, \text{Alice}, -, -, 6, -, \{x \leqslant 6 \leqslant x+7,\ 6 \leqslant z \leqslant 13\}]$ are in $\textsc{ba}_r(\pi_1)$ at $\texttt{ts} = 9$ and agree on u. Their combination $\mathit{merge}(\mu_2, \mu_3)$ satisfies $x \leqslant 6 \leqslant x+7$ and $3 \leqslant y \leqslant 10$, so a row corresponding to $\mathit{merge}(\mu_2, \mu_3)$ is added to $\textsc{ba}_r$ as μ_5. ▌

The following lemma states that Update refreshes the body and head tables by adding exactly the new assignments that can be derived from $\eta \cup \Delta$.

Lemma 4.2 : Let $r{:}\varphi \to \psi$ be a rule, η an enactment, and Δ a batch for η. $\mathsf{Update}(\varphi, \Delta, \textsc{ba}_r(\eta))$ computes $\textsc{ba}_r(\eta \cup \Delta)$ and $\mathsf{Update}(\psi, \Delta, \textsc{ha}_r(\eta))$ computes $\textsc{ha}_r(\eta \cup \Delta)$.

Proof: We show this for $\textsc{ba}_r(\eta \cup \Delta)$; adapting this argument for $\textsc{ha}$ is trivial. We start by assuming $\textsc{ba}_r(\eta)$ contains every assignment μ such that for a non-empty subset P of the event atoms in φ, μ is defined for the variables in P, $\mu(P) \subseteq \eta$, and for each gap atom g whose variables appear in P, η satisfies g with μ. Let μ be an assignment such that for a non-empty subset P of the event atoms in φ, μ is defined for the variables in P, $\mu(P) \subseteq \eta \cup \Delta$, and for each gap atom g whose variables appear in P, $\eta \cup \Delta$ satisfies g with μ. We now show that μ is inserted in $\textsc{ba}_r(\eta \cup \Delta)$ by $\mathsf{Update}(\varphi, \Delta, \textsc{ba}_r(\eta))$.

Because $\mu(P) \subseteq \eta \cup \Delta$, the $\mathcal{D}$ and $\mathcal{T}$ values in μ are derived from some set of events C in η and some set of events D in Δ. We assume that C is non-empty; otherwise, μ is already in $\textsc{ba}_r(\eta)$ by the starting assumption. Additionally, by the starting assumption,

Algorithm 3 Update-E$(\Delta, \textsc{ext}_r(\eta), \textsc{ba}_r(\eta \cup \Delta), \textsc{ha}_r(\eta \cup \Delta))$

```
Input: A batch $\Delta$, un-updated table $\textsc{ext}_r(\eta)$,
       updated tables $\textsc{ba}_r(\eta \cup \Delta)$ and $\textsc{ha}_r(\eta \cup \Delta)$ for an enactment $\eta$
Output: The updated table $\textsc{ext}_r(\eta \cup \Delta)$
 1: $\Gamma := \textsc{ext}_r(\eta)$ ;
 2: for each complete body assignment $\mu$ in $\textsc{ba}_r(\eta \cup \Delta)$ do
 3:   if $\max(\mu) = \mathsf{ts}_\Delta$ then Add $\langle \mu, \text{-}, \mathsf{Deadline}(\psi, var(\psi), \mu) \rangle$ to $\Gamma$ ;
 4:   end if
 5: end for
 6: for each assignment $\gamma$ in $\textsc{ha}_r(\eta \cup \Delta)$ do
 7:   if $\max(\gamma) = \mathsf{ts}_\Delta$ then
 8:     for each row $\langle \mu, \beta, d \rangle$ in $\Gamma$ do
 9:       if $\gamma$ extends $\mu \cup \beta$ and $\gamma(\mu(\psi))$ is satisfiable then
10:         Add $\langle \mu, \gamma, \mathsf{Deadline}(\psi, var(\psi), \mu \cup \gamma) \rangle$ to $\Gamma$ ;
11:         if $\gamma$ is complete then Update $\textsc{ba}_r(\eta \cup \Delta)$ to indicate $\mu$ has a match ;
12:         end if
13:       end if
14:     end for
15:   end if
16: end for
17: output $\Gamma$ ;                                        /* $= \textsc{ext}_r(\eta \cup \Delta)$ */
```

$\textsc{ba}_r(\eta)$ contains an assignment μ_C derived from C. Then, we can show μ is created by merging μ_C with some assignment μ_D for D. The double **for** loop of Update generates an assignment for each event d in D and event atoms in φ with the same event type as d. Next, these assignments merge with each other in the while loop; because $\mu(P) \subseteq \eta \cup \Delta$ and P's gap atoms are consistent with μ, μ_D is consistent with P and created by the while loop, then added to $\textsc{ba}_r(\eta \cup \Delta)$. Alternatively, consider any assignment μ that is not in $\textsc{ba}_r(\eta \cup \Delta)$ after Algorithm 2. Then, no subset of events in $\eta \cup \Delta$ can create μ or μ is inconsistent with φ and will not pass the checks for consistency with $\Theta = \varphi$. ▮

The EXT table pairs body assignments with head assignments; it is updated by Update-E (Alg. 3). When a batch arrives, Update-E (Alg. 3) adds new complete body assignments to EXT (Lines 2-3), and then adds pairs using head assignments (Lines 4-8), computing a deadline for each pair (Line 8). Line 9 checks if there is a match between complete body and some head assignment, updating BA if so. An example is shown in Fig. 4.5 for the complete assignments in Fig. 4.4.

For all complete body assignments, EXT stores each head assignment that extends it and indicates the latest time the head assignment can be further extended. The following lemma characterizes the conditions and time whereby a violation can be detected using EXT.

Lemma 4.3 : Let $r\colon \varphi \rightarrow \psi$ be a rule, η an enactment, τ a timestamp, and μ a complete body assignment for r. Then, μ is a witness of a violation of r in η at τ iff no rows in $\textsc{ext}_r(\eta)$ at τ pairs μ with a complete head assignment or with a deadline for μ greater than τ.

Proof: Let τ be the largest timestamp in η. At time τ, by the definition of EXT, $\textsc{ext}_r(\eta)$ contains all pairs for μ and head assignments from $\textsc{ha}_r(\eta)$, so if μ is unmatched in $\textsc{ba}_r(\eta)$, there is no assignment with $min(\beta) \leqslant \tau$ that extends μ and satisfies ψ. Alternatively, let τ be the largest deadline for μ in $\textsc{ext}_r(\eta)$; then, by Lemma 4.1, for all rows with μ and β in $\textsc{ext}_r(\eta)$, for all complete head assignments β' that extend $\mu \cup \beta$, such that $\max(\beta'(U)) > \tau$, $\beta'(\psi)$ is inconsistent. Thus, no future (i.e., with a value greater than τ) complete head assignment can extend μ and satisfy ψ. Then, μ will never be extended by a complete head assignment that satisfies ψ, so μ is a witness for a violation for η. ▮

Example 4.4 : In Section 2.3, μ_{10} satisfies φ and must be extended no later than 9. Then, the deadline for matching the unpaired μ_{10} in $\textsc{ext}_r(\eta_{\leqslant 9})$ is 9. At time 9, a `Payment` event creates β_2 (Fig. 4.4), and μ_{10} and β_2 are inserted into $\textsc{ext}_r(\eta_{\leqslant 9})$ with deadline 13 because $\beta_2(w) = 9$ and ψ contains $v \leqslant w + 4$. Assuming no matching `Launch` event arrives, μ_{10} can be reported as a violation at time 13. ▮

We now present the algorithm Detect (Algorithm 4) that detects violations. These are unmatched body assignments in EXT (1) whose largest deadline is less than or equal to the current time or (2) whose enactments have ended.

Finally, we state that Detect yields all violations of rules at the earliest possible time.

Theorem 4.4 : Let $r\colon \varphi \rightarrow \psi$ be a rule, η an enactment, and Δ a batch. Then, μ is a witness of a violation in $\eta \cup \Delta$ iff $\mathsf{Detect}(Delta, \textsc{ext}_r(\eta \cup \Delta))$ contains μ.

Algorithm 4 Detect$(\Delta, \textsc{ext}_r(\eta \cup \Delta))$

Input: A batch Δ, the table $\textsc{ext}_r(\eta \cup \Delta)$

Output: A set of assignments indicating rule violations

```
Violations := {};
for each complete body assignment $\mu$ in $\textsc{ext}_r(\eta \cup \Delta)$ do
  if $\mu$ is not extended by some complete head assignment then
    if $\Delta$ contains an END event $e$ with $e.\textsc{id} = \mu.\textsc{id}$ then
      Add $\mu$ to Violations ;
    end if
    Let $\tau$ be the maximum deadline for the rows in $\textsc{ext}_r(\eta \cup \Delta)$ with $\mu$;
    if $\mathsf{ts}_\Delta \geqslant \tau$ then
      Add $\mu$ to Violations ;
    end if
  end if
end for
output Violations ;
```

Proof: We show this by proving both directions of the equivalence. First, assume μ is a violation in $\eta \cup \Delta$. By definition, for all extensions E of $\eta \cup \Delta$, there is no complete head assignment β that extends μ such that E satisfies ψ with β. There are two cases of $\eta \cup \Delta$, depending on whether μ is complete in $\eta \cup \Delta$. $\eta \cup \Delta$ may contain an *END* event and will have no later events, in which case, η.*END* is in Δ. Because μ has no complete matching head assignment for any extension, it is not extended by a complete head assignment in $\textsc{ext}_r(\eta \cup \Delta)$. Then, μ will be added to *Violations*. Otherwise, by Lemma 4.3, μ is complete and in all rows in $\textsc{ext}_r(\eta \cup \Delta)$, it appears with a deadline of, at most, ts_Δ. Then, the condition $\mathsf{ts}_\Delta \geqslant \tau$ is satisfied, and μ is be added to *Violations*.

Conversely, assume μ is in Detect$(\Delta, \textsc{ext}_r(\eta \cup \Delta))$. Then, μ is added to *Violations* somewhere in Detect with inputs Δ and $\textsc{ext}_r(\eta \cup \Delta)$. *Violations* is extended only within the **for** loop in Detect, which iterates over complete body assignments, so μ is a complete assignment in $\textsc{ext}_r(\eta \cup \Delta)$. The algorithm also checks whether μ is extended by a complete head assignment, so μ is not extended by any complete head assignment. Then, there are two cases for μ to be added to *Violations*: either (1) Δ contains η.*END* or (2) ts_Δ is greater than or equal to all deadlines for rows with μ in $\textsc{ext}_r(\eta \cup \Delta)$. If (1) holds, then μ is a violation because $\eta \cup \Delta$ will have no later events, so no future complete head assignment will be created for $\eta \cup \Delta$, so no future complete head assignment matching μ

will satisfy ψ. Then, μ is a witness of a violation in $\eta \cup \Delta$. If (2) holds, by Lemma:4.3, so no future complete head assignment matching μ will satisfy ψ. Then, μ is a witness of a violation in $\eta \cup \Delta$. In either case, μ is a witness of a violation in $\eta \cup \Delta$. ▮

Theorem 4.4 indicates that our techniques reports violations at the earliest possible time. This concludes the data structures and algorithms for individual rules.

4.2 Algorithms for an Acyclic Set of Rules

In this section, we present an algorithm to detect violations of a given set of rules. We first demonstrate how conflicts between rules can lead to violations that the algorithm for individual rules in the previous section cannot detect early. A key step in the algorithm here is the "chase" process, which generates expected events to aid reasoning about rule violations by instantiating head event atoms for unmatched complete body assignments. The chase process requires augment data structures and extended update algorithms. An obstacle here is that the chase process may sometimes not terminate. We define a subclass of "acyclic" sets of rules that guarantees termination of the chase process, and restrict our results to acyclic sets of rules.

Recall that in the case of multiple rules, the notion of rule violation states that an enactment η constitutes a violation of a set of rules R if in each (complete) extension of η, there is a violation for some rule in R. Notably, unlike the single-rule case, the presence of a violation may not imply that any particular rule is violated or a witness exists, only that some rule is or will violated in all extensions of the enactment. This demands a different approach to detecting violations, as we cannot simply apply single-rule violation detection to each rule separately. The following example illustrates how reasoning about individual rules may not reveal a violation of a rule set.

Example 4.5 : Consider the following two rules:

$$R_1 : \texttt{Request}@x \rightarrow \texttt{Schedule}@y, x+1 \leqslant y \leqslant x+2$$

$$R_2 : \texttt{Request}@x, \texttt{Schedule}@y, x+2 = y \rightarrow \texttt{Payment}@z, x = z$$

and the enactment $\eta = \{\texttt{Request}@10, \texttt{Payment}@11\}$. We assume no other events with timestamps less than 12 will be added to η. Considering R_1 alone, the earliest time to detect a violation of R_1 in η is 12 because we have $y \leqslant 10+2 = 12$ in R_1 for $x \mapsto 10$. Considering R_2 alone, that there is no violation of R_2 because there is no `Schedule` event in η. However, to satisfy R_1 with the mapping $x \mapsto 10$, there must be a `Schedule` event at time 11 or 12; otherwise, there is a violation of R_1 at time 12. Given that η has no `Schedule` event at time 11, the `Schedule` event must be at time 12 to satisfy R_1. Then, to satisfy R_2 for the mapping $x \mapsto 10$, $y \mapsto 12$, there must be a `Payment` event at time 10, which is not in η. Then, all extensions of η violate either R_1 or R_2, so η violates $\{R_1, R_2\}$ at time 11. This violation is inevitable at time 11, despite the fact that considering R_1 and R_2 individually deduces a deadline of 12 for a violation. ▮

To detect violations for sets of rules, we observe that the head events required to avoid violations may satisfy a rule body in the future, which may trigger rules to require other head events. To formalize this reasoning, we use a technique for reasoning about constraints on relational databases: a chase [48]. In our setting, the chase generates "expected" events: events that are expected to occur to avoid violations. This happens when a rule's body is satisfied and no corresponding head assignment exists. Then, the rule is applied by instantiating the head's existential variables with "marked nulls", placeholders for unknown time instants or values, marked to distinguish different values and time instants. These marked nulls then ground the head event atoms to create expected events, which are then processed as if they were "real" events in the enactment.

Example 4.6 : Continuing with Example 4.5, we apply a chase process to η and $\{R_1, R_2\}$.

Applying R_1 to Request@10 yields an expected event Schedule@y_1 with marked null y_1 and constraints $11 \leqslant y_1 \leqslant 12$. Assuming $y_1 = 12$ and applying R_2 to Schedule@y_1 yields an expected event Payment@z_1 with marked null z_1 and constraint $z_1 = 10$, i.e., Payment@10 is expected if $y_1 = 12$. Note that applying R_1 generates an expected event, which is then used to apply R_2. Given $11 \leqslant y_1 \leqslant 12$ and the current time 11 for η, we conclude $y_1 = 12$. The necessary Payment event is not in η, so η violates $\{R_1, R_2\}$ at time 11. ▮

We adapt the data structures and algorithms from the previous section to support the chase. Recall that the algorithms for individual rules used three tables for a rule r: BA_r for body assignments, HA_r for head assignments, and EXT_r (extensions) to track pairings of body and head assignments. For the multi-rule case, these tables are used with two changes. First, BA_r is augmented with an additional column named *Chased* (using values *yes* and *no*). to indicate if an assignment has been chased and thus should not be chased again. The columns of the HA_r and EXT_r tables are unchanged. Second, these three tables may have marked nulls to represent unknown time instants or values generated by the chase. To denote the inputs and outputs of the chase, we define an *assignment database* $D_R(\eta)$ that consists of the following tables: for each rule r in R, $\text{BA}_r(\eta)$ (with the additional *Chased* column), $\text{HA}_r(\eta)$, and $\text{EXT}_r(\eta)$.

For an enactment η and a set of rules R, the chase takes as inputs $D_R(\eta)$ and a batch Δ, and produces as the output the updated assignment database $D_R(\eta \cup \Delta)$ (if it terminates). The chase may not terminate, an issue we discuss after presenting the algorithm.

The chase shown as Algorithm 5. It computes an extended assignment database for the enactment $\eta \cup \Delta$ and rules R given a batch Δ of new events for η, rules R, and the assignment database $D_R(\eta)$. First, the algorithm checks for unmatched complete body assignments; if not, the algorithm terminates on Line 1 with the assignment database

Algorithm 5 Chase$(\Delta, D_R(\eta))$

```
Input: A batch $\Delta$ of events for $\eta$, rules $R$,
       the assignment database $D_R(\eta)$,
Output: the assignment database $D_R(\eta \cup \Delta)$
 1: if $D_R(\eta)$ contains no unmatched complete body assignments then
 2:   return $D_R(\eta)$
 3: end if
 4: Let $\mathit{ExpectedEvents} := \Delta$
 5: while $\mathit{ExpectedEvents}$ is not empty do
 6:   Update all $\text{BA}_r(\eta), \text{HA}_r(\eta)$ tables using Update and $\mathit{ExpectedEvents}$ as the batch
 7:   Update all $\text{EXT}_r(\eta)$ tables using Update-E and $\mathit{ExpectedEvents}$ as the batch
 8:   Let $\mathit{ExpectedEvents} := \varnothing$
 9:   for each complete, unchased body assignment $\mu \in D_R(\eta)$ do
10:     if $\mu$ has no ground matching head assignment for a rule $r$ then
11:       Let $h$ be a mapping from each existential variable in $\mathit{head}(r)$ to a fresh marked null
12:       for each event atom $a$ in $\mathit{head}(r)$ do
13:         Let $\beta$ be the gap atoms in $\mathit{head}(r)$
14:         Add $(a, (\mu \cup h)(\beta))$ to $\mathit{ExpectedEvents}$
15:       end for
16:     end if
17:     Modify the $\mathit{Chased}$ column of $\mu$ to $\mathit{yes}$
18:   end for
19: end while
20: return $D_R(\eta)$
```

unchanged. The assignment database is updated on Lines 3-4 as in the single-rule case with the batch Δ. We use the same update algorithms as in the single-rule case, though Update is modified to account for the *Chased* column in the body table (initialized to *no* for new assignments) and Update-E is modified to treat marked nulls timestamps as unresolved variables in deadline calculation and to only mark body assignments as matched with *ground* head assignments. Then, the algorithm chases the unmatched complete body assignments on Lines 7-12. Each such assignment μ for a rule r is chased by creating a mapping h from the existential variables to fresh marked nulls on Line 8. Then, for each event atom α in the head of r, the algorithm creates an expected event with h applied to α and the gap atoms in the head of r on Lines 9-11. On Line 12, the row for μ in $\text{BA}_r(\eta)$ is modified to prevent μ from being chased again. Because expected events may generate more expected events, the **while** loop is only exited if no new expected events are created from the last group of unmatched, complete body assignments.

There is an issue with the chase's termination; in general, it may not terminate, as marked nulls can create more marked nulls, ad infinitum. As a trivial example, consider the rule R : $\texttt{A}@x \rightarrow \texttt{A}@y, x + 1 = y$; the chase will generate an infinite number of expected events. We enforce chase termination by considering only "acyclic" sets of rules. Intuitively, acyclicity requires that a marked null cannot create another marked null for the same attribute in an event schema. We use the following definition of acyclicity, derived from the weak acyclicity property in [49]that ensures that chase termination.

Definition : Let R be a set of rules, the graph $G_R = (V, E)$ is defined as follows.

- V is a set of vertices (P, a) where P is an event name and a is an attribute of P,
- E is a set of edges For every rule $\phi(\bar{x}) \rightarrow \psi(\bar{x}, \bar{y})$ in R, we call each x in $\bar{x}$ a propagated variable. For each propagated variable x, for each occurrence of x in $\phi(\bar{x})$ in position (P, a), do two things:
  1. for each occurrence of x in $\psi(\bar{x}, \bar{y})$ at position (Q, b), add an edge from (P, a) to (Q, b) (for some event name Q and attribute b),
  2. for each occurrence of an existentially quantified variable y in $\psi(\bar{x}, \bar{y})$, for each occurrence of y in $\psi(\bar{x}, \bar{y})$ at position (S, c) (event name S with attribute c), add a *special edge* from (P, a) to (S, c)

A set of rules R is *acyclic* if G_R has no cycle containing at least one special edge.

We now discuss a property of Chase and the assignment database it produces. Recall that the chase generates expected events when it encounters an unmatched, complete body assignment μ for a rule r. For μ to be satisfied, a ground version of each expected event must eventually appear; otherwise, r is not satisfied with respect to μ. If r is chased with μ yielding expected events E, then each grounding of an assignment database $D_R(\eta)$ with no violations for R has a match for μ; this match indicates there is a grounding for each expected event in E. We formalize this in the following lemma.

Lemma 4.5 : Let R be a set of rules, η an enactment, and $D_R(\eta)$ the chased assignment database. Then, there is an assignment h to the marked nulls in $D_R(\eta)$ such that $h(D_R(\eta))$ has no violations if and only if there is a set of ground events H such that $\eta \cup H$ satisfies R.

Proof: We prove the lemma by showing both directions of the equivalence. First, assume there is an assignment h to the marked nulls in $D_R(\eta)$ such that $h(D_R(\eta))$ has no violations. Let E be the set of expected events created by $\mathsf{Chase}(D_R(\eta))$. Because all marked nulls in $D_R(\eta)$ are assigned by h, $H = h(E)$ is a set of ground events. Let μ be a complete body assignment in $D_R(\eta)$ for some rule r. If μ is matched by a head assignment, it is was not chased. Otherwise, chasing μ created some head assignment β in $D_R(\eta)$ and some expected events $E' \subseteq \beta(\mathit{head}(r))$ in E. Because h grounds $D_R(\eta)$, h grounds β and thus E'. Then, $\eta \cup h(E')$ satisfies r with respect to μ with the match $h(\beta)$. This holds for all complete body assignments in $D_R(\eta)$, so $\eta \cup h(E)$ satisfies R.

Next, assume there is a set of ground events H such that $\eta \cup H$ satisfies R. Let μ be a complete body assignment in $D_R(\eta)$ for some rule r. If μ is matched by a head assignment β in $D_R(\eta)$, it is not a violation. Otherwise, μ is chased to create some head assignment β in $D_R(\eta)$ with marked nulls. Because $\eta \cup H$ satisfies R, $\eta \cup H$ satisfies r with respect to μ; let γ be the head assignment in $\eta \cup H$ that matches μ. Then, γ grounds the marked nulls in β. Thus, γ is a ground head assignment in $D_R(\eta)$ for r and μ. Let h map each marked null in $D_R(\eta)$ created by chasing some μ to the values in a head assignment that matches μ in $\eta \cup H$. Then h grounds each assignment created by chasing μ. Then, $h(D_R(\eta))$ has no violations. ▮

We now discuss the violation detection algorithm for acyclic sets of rules. We define a new algorithm Detect-Multi Recall that Detect (Algorithm 4) creates a formula $\mathsf{ts}_\Delta \geqslant \tau$ with the current time ts_Δ and the assignment's largest deadline τ, then tests this formula's satisfiability to determine if the enactment has a violation. Detect-Multi uses a more

Algorithm 6 Build($D_R(\eta)$)

```
Input: An assignment database $D_R(\eta)$, a time instant $t$
Output: a formula $\Theta$
 1: $\Theta := true$
 2: for each complete body assignment $\mu$ in $D_R(\eta)$ with gaps $g_\mu$ and rule $r$ do
 3:     if $\mu$ has no complete matching head assignment then
 4:         Add $\neg g_\mu$ to $\Theta$
 5:     end if
 6:     if $\mu$ has complete matching head assignments $\beta_1, \ldots, \beta_n$ where $(\mu, \beta_i, g_i, t_i)$ are the rows with $\mu$ in $\textsc{ext}_r(\eta)$
        then
 7:         Add $g_\mu \rightarrow (g_1 \vee \cdots \vee g_n)$ to $\Theta$
 8:     end if
 9: end for
10: for each marked null timestamp $x$ do
11:     Add $x > t$ to $\Theta$
12: end for
13: return $\Theta$
```

complex formula than Detect; it uses the Build algorithm (Algorithm 6) to produce a formula Θ from the assignment database for the current enactment η and R such that Θ is unsatisfiable if and only if η violates R. Then, we apply satisfiability testing to Θ; in practice, this is done by calling a SAT solver. In the multi-rule case, there may not be a witness for the violation, as would be reported in Detect. Instead, if the formula is unsatisfiable, Detect-Multi simply reports η as *violating* R, otherwise, *not violating.*

Build (Algorithm 6) starts with the assignment database $D_R(\eta)$ and a time instant t, meant to be the current time. A formula Θ is initialized as *true* on Line 1, as no indication of violations has been found yet. The **for** loop on Line 2 iterates over the complete body assignments μ in $D_R(\eta)$ with (unresolved) gaps g_μ. There are two cases of the extension table for μ. In the first case, covered by Line 3, μ has no complete matching head assignments. Recall that g_μ contains the assumptions made about μ's variables required for μ to satisfy the rule body. Accordingly, if these assumptions are not true, μ does not represent a valid assignment in η; we check if these assumptions can be avoided by adding $\neg g_\mu$ to Θ in Line 4, thus testing if their negation is satisfiable. In the second case, covered by Line 5, μ has one or more complete matching head assignments in its extension table. Let (μ, β_1, g_1, t_1), ..., (μ, β_n, g_n, t_n) be the rows matching μ in its extension table, where β_i is a matching head assignment, g_i is the set of gaps for β_i, and

t_i the deadline for extending β_i. Note that μ is a witness for a violation if every g_i is unsatisfiable and each deadline t_i is less than or equal to the current time t. For the same reasoning as the first case, matching μ is necessary only when g_μ is true. Additionally, only one matching β_i is needed, so only one corresponding sets of gaps g_i is needed when g_μ is satisfied. Thus, $g_\mu \to (g_1 \vee \cdots \vee g_n)$ is added to Θ in Line 6. Finally, the algorithm adds the requirement that each marked null timestamp x is greater than t, the current time, to Θ in Lines 7-8, as marked nulls represent unknown time instants, thus they must be in the future.

Example 4.7 : Consider again the rules from Example 4.5:

$$R_1 : \texttt{Request}@x \to \texttt{Schedule}@y, x + 1 \leqslant y \leqslant x + 2$$

$$R_2 : \texttt{Request}@x, \texttt{Schedule}@y, x + 2 = y \to \texttt{Payment}@z, x = z$$

and the enactment $\eta = \{\texttt{Request}@10, \texttt{Payment}@11\}$ with current time 11. We build Θ for the corresponding assignment table $D_{\{R_1,R_2\}}(\eta)$. The R_1 body assignment $x \mapsto 10$ with no gaps has a matching head assignment $y \mapsto y_1$ with gaps $y_1 = 11 \vee y_1 = 12$, so we add $\mathit{True} \to (y_1{=}11 \vee y_1{=}12)$ to Θ. The R_2 body assignment $x \mapsto 10, y \mapsto y_1$ with gaps $12 = y_1$ has a matching head assignment $z_1 \mapsto 10$ with gap $z_1 = 10$. Accordingly, we enforce $z_1 = 10$ only if $y_1 = 12$, so we add $(y_1 = 12) \to (z_1 = 10)$ to Θ. Finally, the current time is time instant 11 and the assignments have marked nulls y_1 and z_1, so we add $y_1 > 11$ and $z_1 > 11$ to Θ. In summary, Build produces $(\mathit{True} \to (y_1{=}11 \vee y_1{=}12)) \wedge (y_1{=}12 \to z_1{=}10) \wedge (y_1{>}11) \wedge (z_1{>}11)$. Note that this formula is unsatisfiable because $z_1 = 10$ and $z_1 > 11$ are contradictory. ▮

Now we can describe an algorithm Detect-Multi to detect violations of acyclic sets of rules. Specifically, Detect-Multi takes as input the assignment database $D_R(\eta)$ for an acyclic set of rules R and an enactment η, as well as a batch Δ of events for η. Detect-Multi computes the updated assignment table $D_R(\eta \cup \Delta)$ with Update and Update-E to integrate the batch Δ into $D_R(\eta)$. Then, Detect-Multi applies the Chase to compute the expected events and Update and Update-E to extract the corresponding assignments, integrating them into $D_R(\eta)$. Once the assignment table $D_R(\eta)$ is updated, Build compute a formula Θ from $D_R(\eta)$ and checks its satisfiability. If Θ is unsatisfiable, then Detect-Multi reports a violation.

It is desirable to establish that this method of detecting violations is sound and complete; we do so in Theorem 4.6. The crux of the theorem comes from showing that Build's formula Θ mirrors the conditions for a violation. Build gathers these conditions using complete body assignments and matching head assignments, which have marked nulls for unresolved variables. Accordingly, there is a viable choice of timestamps and values for the marked nulls and unresolved variables in Θ if and only if there is some extension of the current enactment with no violations. We establish Theorem 4.6 by showing that Θ is unsatisfiable if and only if η violate R.

Theorem 4.6 : Let R be an acyclic set of rules and η an enactment. Then, Build$(D_R(\eta))$ is unsatisfiable iff η violates R.

Proof: First, we show that if Build$(D_R(\eta))$ is satisfiable, then η doesn't violate R, i.e., there is an extension of η that satisfies R. Let Θ be the formula returned by Build$(D_R(\eta))$. We assume Θ is satisfiable; let h be a satisfying assignment for Θ. Note that the variables in Θ are the marked nulls in $D_R(\eta)$; we use this satisfying assignment to show $h(D_R(\eta))$ is a ground assignment database with no violations of R. Let μ be an arbitrary, complete body assignment in $D_R(\eta)$ for some rule $r \in R$. We show that $h(\mu)$ does not witness a violation of r in $h(D_R(\eta))$. Let g_μ be the gap for μ's body and $g_1, \ldots, g_n$ the gaps in head assignments $\beta_1, \ldots, \beta_n$ extending μ in $\textsc{ext}_r(\eta)$. Then, $g_\mu \rightarrow (g_1 \vee \cdots \vee g_n)$ is a clause in Θ, Because h satisfies Θ, some g_i is consistent with h; then, the head assignment β_i can be extended with h to a complete head assignment matching μ in $h(D_R(\eta))$. Alternatively, it may be that μ has no matching head assignment in $\textsc{ext}_r(\eta)$;

then, $\neg g_\mu$ is a clause in Θ. Because h satisfies Θ, h satisfies $\neg g_\mu$, so the conditions that ground μ are not satisfied in $h(D_R(\eta))$. Then, μ is not a witness of r in $h(D_R(\eta))$. We have shown an arbitrary complete body assignment in $h(D_R(\eta))$ is not a witness; then, $D_R(\eta)$ is an assignment database with no violations of R. Applying Lemma 4.5, there is a set of events H derived from h such that $\eta \cup H$ that satisfies R. By the construction of Θ, h maps each marked null timestamp to a timestamp greater than the current time, so each event in H is in the future of the current time, making H a batch of future events for η. Then, $\eta \cup H$ is an extension of η that satisfies R, so η doesn't violate R.

Alternatively, assume that Θ is unsatisfiable. Then, there is no satisfying assignment that satisfies Θ. Equivalently, for each assignment h for Θ, h does not satisfy Θ. Let h be an arbitrary assignment for Θ. Because h does not satisfy Θ, one of the following holds: (1) h assigns some marked null timestamp a value less than the current time, (2) h does not satisfy some clause $\neg g_\mu \rightarrow$ in Θ for some complete body assignment μ of some $r \in R$ in η, or (3) h does not satisfy some clause $g_\mu \rightarrow (g_1 \vee \cdots \vee g_n)$ in Θ for some complete body assignment μ of some $r \in R$ in η. In the first case, some timestamp in h is in the past of the current time, so h cannot lead to an extension of η. Alternatively, we show that if (2) or (3) holds, $h(D_R(\eta))$ does not satisfy R. If (2) holds, for some complete body assignment μ in $h(D_R(\eta))$, $\neg g_\mu$ is not satisfied; then, μ is a ground complete body assignment with no head assignment matching μ. Because $D_R(\eta)$ has been chased, μ having no matching head assignment indicates $\mu(\mathit{head}(r))$ is inconsistent. Then, μ is a witness of a violation in $h(D_R(\eta))$, If (3) holds, for some complete body assignment μ in η, $h(g_\mu)$ is true and $h(g_1 \vee \cdots \vee g_n)$ is false. Then, no head assignment matching μ in η can be extended to a complete head assignment in $h(D_R(\eta))$, so μ is a witness of a violation. In both cases, $h(D_R(\eta))$ has a violation. Then, for all assignments h to the marked nulls in $D_R(\eta)$, $h(D_R(\eta))$ has a violation. By Lemma 4.5, there is no set of events H derived from h such that $\eta \cup H$ that satisfies R. Then, there is no extension of η that satisfies R. Then, η violates R. ∎

From Theorem 4.6, we see that our techniques reports violations at the earliest possible time. This concludes the data structures and algorithms for sets of rules.

4.3 Optimizations

While the algorithms presented in Section 4.1 report violations correctly, their time and space complexities can be improved. We present two optimizations: one to remove useless assignments using a similar reasoning to deadline calculation and one to avoid repeated computation by tracking which data is new. We report their improvement of relevant algorithms as a factor of the log size $|L|$, the batch size $|\Delta|$, the number of active enactments as approximated by $|\Delta|$, and the number of event atoms in the rule body or head e.

Expiring partial assignments: Early violation detection motivates a similar technique for discarding useless assignments. Assignments in BA and HA are *expired* (i.e., useless) if (1) they can no longer be extended because their timestamps and unresolved gap atoms are inconsistent with all possible future assignments, or (2) they are derived from an enactment that has ended. It is much desired to remove expired assignments, and thus reduce the sizes of BA and HA. Calculating expiration times resembles deadline calculation; in fact, the Deadline function is reused. To incorporate expiration time, we augment BA and HA (resp.) with an *expiration* column as new tables BAE and HAE, requiring that incomplete assignments in BAE and HAE be extendable by future events to complete assignments. To maintain this property, Deadline calculates its expiration time for each incomplete assignment with respect to its unresolved gap atoms. Removing expired assignments reduces the size of the BAE and HAE tables from $O(|L|^e)$ to $O(|\Delta|^e)$, which benefits the algorithms in Section 4.1 by reducing the number of computations in Update from $O(|L|^{2e})$ to $O(|\Delta|^{2e})$, and that in Update-E from $O(|L|^e)$ to $O(|\Delta|^e)$. It also improves Update-E by decreasing the number of assignments checked for insertion into

EXT (Lines 2 and 4), from $O(|L|^e)$ to $O(|\Delta|^e)$.

Semi-naive MERGE of assignments: We can also decrease the number of computations in the Update algorithm by tracking which data generated by the most recent batch. The **while** loop (Lines 7-10) in Update tests pairs of assignments to merge. For each batch Δ, we only need to try pairs that have at least one assignment added from events in Δ, because all other pairs of assignments were considered before Δ arrived. To make Update to reflect this, we use a queue Γ_{new} to hold new assignments generated at Line 6. We exchange the **for** loop in Update (Lines 8-10) to a doubly nested **for** loop that iterates through each assignment μ_n in Γ_{new} (outer loop) and each row μ_o in Γ (inner loop), adding the new assignment to the queue Γ_{new}, moving μ_n from Γ_{new} to Γ after processing μ_n. This resembles "semi-naive" evaluation of Datalog programs [48] and reduces the search for matching assignments from considering $O(|L|^{2e}|)$ pairs to only pairs involving some new data: $O(|L|^e|\Delta|^e)$ pairs.

4.4 Evaluation

To assess the advantages and performance of our approach, we conducted evaluations on two types of monitors: a single-rule monitor and a multi-rule monitor. These monitors were constructed using the algorithms outlined in Sections 4.1 and 4.2, respectively, along with optimizations described in Section 4.3. This section focuses on addressing the following key questions: (1) How early can violations be detected, and (2) What are the feasible dimensions of logs and rules for monitoring? We find that single and multi-rule monitoring is effective for logs from medium-sized applications, that is, those with thousands of events per second and hundreds of concurrent users or processes. We also report multi-rule monitoring remains feasible for complex rules, as a function of the number and size of rules, as long as degree of overlap between rules is not too high.

How early violations are detected is crucial as it indicates the percentage of events observed between the first violation and the end of the enactment. Knowing this helps in understanding how much of the enactment's execution occurs while it is still in violation which is particularly important for applications that can halt or mitigate violating enactments. In Section 4.4.2, we study the earliness of violation detection for the single-rule monitor, reporting the average percentage of events observed in violating enactments before and after their first reported violation, for both normal-length enactments (≈10 events per enactment) and large enactments (≈100 events per enactment).

Another critical aspect of evaluation is the feasibility of monitoring with respect to the log and rule properties. Software monitors need time to process each batch of events. When the average batch processing time surpasses the batch arrival rate, the monitor accumulates a backlog of events and does not report violations in real-time. In Section 4.4.3, we evaluate the effect of the batch size and enactment concurrency which approximate the scale and complexity of source systems. The batch size represents the rate of event generation and concurrency is the average number of active enactments over all timestamps. Additionally, different applications may apply different sets of rules. The multi-rule algorithm becomes crucial when rules interact with each other. Hence, we investigate the effects of varying the "overlap" within sets of rules. In Section 4.4.4, we report that multi-rule monitoring is feasible for sets of rules with up to four medium-sized rules and an average overlap per rule pair of up to one event atom.

4.4.1 Generating Logs, Rules, and Experimental Setup

Violation detection may be applied to a service system to monitor multiple enactments simultaneously. A *log* is a sequence of events from multiple enactments, ordered by their timestamps and distinguished by their enactment IDs. Because we define rule violation as occurring within individual enactments and events from different enactments can be distinguished by their IDs, the algorithm development in Sections 4.1 through 4.3 focuses on *when* to detect violations, rather than filtering events by enactment and applying separate monitors to each enactment. There are approaches for separating events from different enactments in a log algorithmically, e.g., trace slicing [50], which filters event data into different monitors based on the enactment data and the rule being applied. Such techniques are orthogonal to our work, and could be used in conjunction with our algorithms to improve performance. We do, however, use logs in the empirical evaluation (Section 4.4), as workflow enactments are often available in an application setting in the form of logs.

To conduct the evaluations, we generated logs of events by simulating customer support workflow models with a workflow simulator [51]. The simulator instantiates the enactment of a workflow with a `Start` event, then advances the enactment based on a probability distribution of activity durations and a simulated resource availability. When an activity is completed, an event with the workflow's id, the activity's name, relevant data attributes, and a timestamp is created. The three workflow models used by the simulator are Ticket Triage, Support Case Processing, and Trust Case Processing, shown in Figure 4.6. Each enactment has five activities and gates that determine the next activity based on some data attribute (e.g., the support level of a customer). Additionally, each enactment begins with a `Start` event and ends with an `End` event.

Logs are created by interleaving events from multiple enactments. We created nine logs with 2,000 enactments and 10,000 events each, achieving different batch sizes and concurrency by changing simulation parameters and post-processing. A portion of one log with a batch size of ten and four concurrent enactments is shown in Figure 4.7. Sample logs are available on Github [52].

We generated rules using the workflows' activities, e.g., `ReadCase`, `SendResponse`, etc., and gap atoms with one- to ten-second gaps, reflecting the typical gaps between events in enactments. We respect the workflow model's activity ordering to ensure gap atoms are satisfiable by some enactments. We use two classes of rules: "small" and

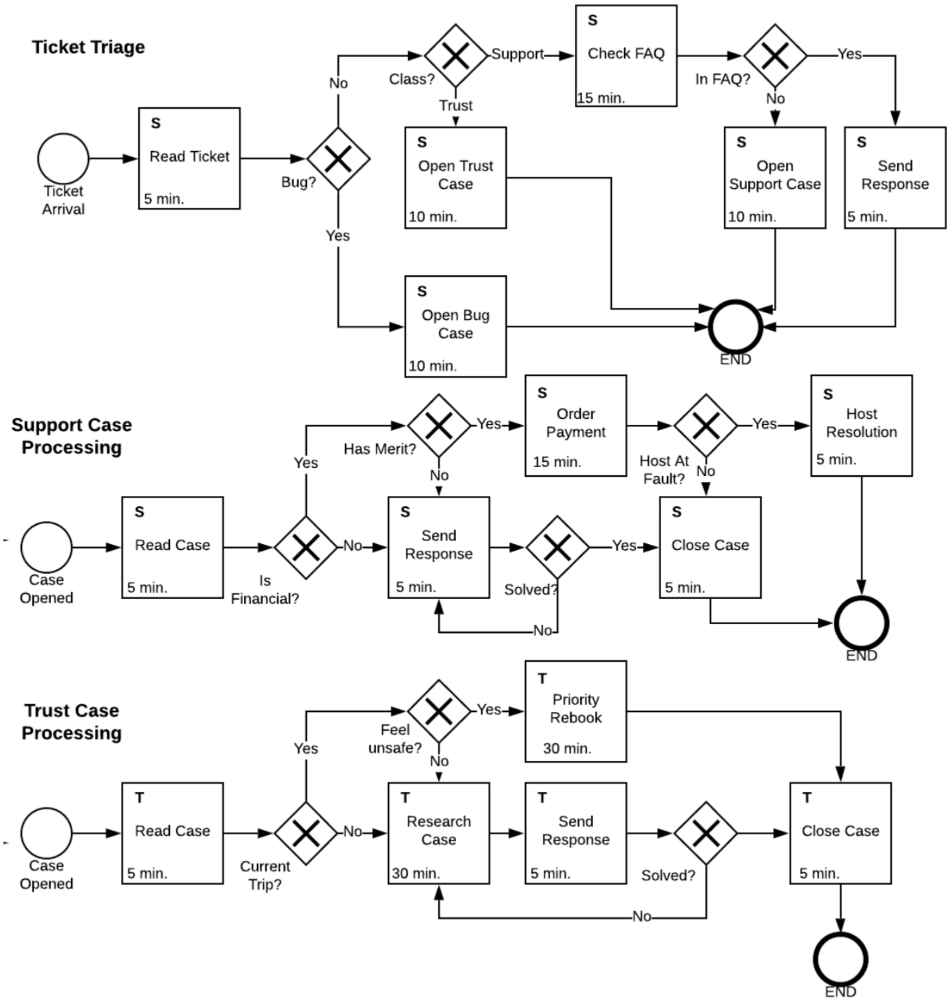

Figure 4.6: Customer support workflows, enacted to generate logs

batch	timestamp	enactment id	event type	event data
42	42	μ_1	ReadCase	{ name=Alice, support=1 }
	42	μ_1	SendResponse	{ name=Alice }
	42	μ_2	ReadCase	{ name=Bob, support=2 }
	42	μ_1	SendResponse	{ name=Alice }
	42	μ_2	SendResponse	{ name=Bob }
	42	μ_2	CloseCase	{ name=Bob, support=2 }
	42	μ_1	CloseCase	{ name=Alice, support=1 }
	42	μ_3	ReadCase	{ name=David, support=2}
	42	μ_4	ReadCase	{ name=Charlie, support=1 }
	42	μ_4	OrderPayment	{ name=Charlie }
43	43	μ_3	SendResponse	{ name=David }
	43	μ_4	SendResponse	{ name=Charlie }
	...	...	...	...

Figure 4.7: Portion of a log with support case enactments with ten events per batch

"medium" rules, as shown in Figure 4.8. For *small* rules, each rule body has one or two event atoms and no more than one gap, and each rule head has one event and no more than one gap. For *medium* rules, each rule body has up to two event atoms and no more than two gaps and each rule head has up to two event atoms and no more than two gaps. We also categorize sets of rules by their "overlap", defined in Section 4.4.4 to approximate the potential for rule interaction.

Small rules (2 to 4 atoms), overlap of 0.33

$$\begin{aligned}
&r_1 : \texttt{ReadCase}(u, i)@x && \rightarrow \texttt{SendResponse}(u)@y, y \leqslant x + 5 \\
&r_2 : \texttt{ReadCase}(u, i)@x, \texttt{SendResponse}(u)@y, && \rightarrow x + 10 \leqslant y \\
&r_3 : \texttt{ReadCase}(u, i)@x, \texttt{OrderPayment}(u)@y, && \rightarrow y \leqslant x + 3, \texttt{CloseCase}(u, i)@z
\end{aligned}$$

Medium rules (5 to 7 atoms), overlap of 0.66

$$\begin{aligned}
&r_4 : \texttt{ReadCase}(u, i)@x && \rightarrow \texttt{SendResponse}(u)@y, x + 3 \leqslant y \\
& && \quad\ \texttt{CloseCase}(u, i)@z, z \leqslant x + 5, z \leqslant y + 1 \\
&r_5 : \texttt{ReadCase}(u, i)@x, \texttt{SendResponse}(u)@y, && \rightarrow \texttt{CloseCase}(u, i)@z, z \leqslant x + 5, z \leqslant y + 1 \\
&\quad x + 4 \leqslant y && \\
&r_6 : \texttt{ReadCase}(u, i)@x, \texttt{OrderPayment}(u)@y, && \rightarrow \texttt{HostResolution}(u)@w, x + 10 \leqslant w \\
&\quad x + 10 \leqslant y && \quad\ y \leqslant w + 10
\end{aligned}$$

Figure 4.8: Sample small and medium-sized rules

Experiments were run on a single machine: a desktop Fedora 20 (Linux) machine with a 2K MHz, 8-core AMD EPYC 7702 processor with 8GB memory. The implementation was written in Python 3.9.6. We used the Z3 SMT solver [53] for satisfiability checking and the Python bindings for Z3 [54]. Notably, in early implementations of the monitor, we invoked the Z3 library many times to instantiate each subformula, then combined these subformula for the satisfiability test with z3.Solver.check. The multiple library calls resulted in much slower processing, up to 100 times as long as our reported results; a frugal use of the solver is critical to achieve reasonable performance. The source code of the monitor and experimental framework is available on Github [52].

4.4.2 Detection Often Occurs Far Before Enactments End

First, we examine how early violations are detected with respect to enactments' events. The quantification of earliness indicates potential benefits where the earliness affects the system's response. For example, the system may choose to halt an enactment when a violation is detected, allowing the system to reclaim resources or to prevent further policy violations. We report the average earliness of violation detection for the single-rule monitor with respect to the enactment's length. Because the multi-rule monitoring algorithm is an extension of the single-rule algorithm and can detect some violations earlier, the earliness afforded by multi-rule monitoring is at least as good as that of single-rule monitoring and may be better in some cases.

We apply the single-rule monitor to logs with normal-length (≈10 events) and large enactments (≈100 events). For each enactment, we count the observed events after the first detected violation. Fig. 4.9 shows that, on average, violations are detected at 75% of event arrival for normal-length enactments and 34% for large ones. As detailed in Section 2.3, this happens when expected head event atoms are pending, but their timestamps are bounded by body event atoms and some gap atoms. Often, the upper bound timestamp of a head event atom falls within the enactment's duration, leading Detect to recognize its occurrence in real-time. We focus on average earliness for single-rule monitoring, but the multi-rule approach often detects even earlier, as shown in Section 4.2. Finally, we note that the average earliness is affected by the gaps in gap atoms; more study is needed to understand the relationship between gap size and average earliness.

	Normal-length enactments		**Large enactments**	
Rule Size	% events before first violation	% after	% events before first violation	% after
Small	74.9	25.1	33.5	66.5
Medium	83.7	16.3	69.4	30.6

Figure 4.9: Percentages of events observed before and after the first detected violation

4.4.3 Detection is Feasible for Medium-scale Applications

Next, we study the feasibility of monitoring. We evaluate when the monitor processes batches in an average of less than one second, the batch arrival rate. The batch size and the enactment concurrency determine the number of assignments that must be joined and matched in the Update and Update-E algorithms, the number of expected events generated by the Chase algorithm, and the number of matches that contribute to the formula produced by Build, so we expect that increasing these parameters raises the processing time.

First, we report the average processing time for the single-rule monitor (Section 4.1) in Figure 4.10. The average processing time is far less than one second for all batch sizes and enactment lengths, shorter than 0.07 seconds for all small rules and shorter than 0.2 seconds for all medium rules, indicating that the single-rule monitor is feasible for logs with those properties. These times increase linearly with the batch size, which is expected for small rules, as the number of possible assignments does not yet suffer the combinatorial explosion. For medium rules, however, the average processing time also increases linearly with the batch size; this is unexpected because to make the number of assignments is exponential in the batch size. This may be due to the increasing number of gap atoms, as more gap atoms decrease the number of assignments in the body or head table. This phenomenon requires a more thorough investigation of the effects of gap atoms on the assignment database.

We also evaluated the feasibility of the multiple-rule monitor. We monitored logs with normal-length enactments and rule sets with three medium rules with an average overlap of one event atom. The results in Table 4.1 show the batch processing time increases with

	Enactment Length			
	Normal	**Large**	**Normal**	**Large**
Batch Size	**Small Rules**		**Medium Rules**	
100	4.55×10^{-4}	6.19×10^{-4}	7.74×10^{-4}	1.363×10^{-3}
1,000	4.330×10^{-3}	6.177×10^{-3}	7.534×10^{-3}	1.3509×10^{-2}
10,000	4.2414×10^{-2}	6.0769×10^{-2}	7.4925×10^{-2}	1.35218×10^{-1}

Figure 4.10: Batch Processing Times (seconds) by Batch Size, Rule Size, and Enactment Length

the batch size and concurrency as expected. Second, the average processing time is ⩽1 second for batches of 100 events and an average enactment concurrency of 100. Beyond these values, the average processing time exceeds ⩾1 second, shown by parentheses in the table, but is still ⩽10 seconds.

We note here that our experiments were conducted on a single commodity machine rather than enterprise-grade hardware; our results suggest that multi-rule monitoring would be feasible for a wider range of applications, i.e., larger batches and more concurrent enactments, if the monitoring is performed with more powerful hardware resources.

Batch Size	**Conc=10**	**Conc=50**	**Conc=100**	**Conc=500**	**Conc=1,000**
10	0.061	0.046	0.294	(1.253)	(3.031)
50	0.195	0.169	0.403	(2.506)	(3.269)
100	0.374	0.328	0.553	(2.807)	(3.483)
500	(1.256)	(1.851)	(1.326)	(3.347)	(5.279)

Table 4.1: Average Batch Processing Time (seconds) by Batch Size and Concurrency

4.4.4 Detection Remains Feasible for Complex Sets of Rules

Another factor affecting the feasibility of monitoring is the size and "overlap" the sets of rules. The *overlap* for a set of rules is the number of pairs of event atoms that share the same name such that one appears in the head of one rule and the other appears in the body of another rule, divided by the total number of rule pairs. More overlap between rules increases the number of assignments generated by expected events from the Chase algorithm, thus increasing the number of assignments to process into the assignment database by Update and Update-E. Furthermore, expected events can create more expected events in the subsequent executions of the Chase algorithm's **while** loop, but this feedback is ultimately limited by the acyclicity of the rule set. Finally, expected events carry marked nulls, which are added to the formula produced by Build, so more overlap grows the subformula in the formula that share variables; this may increase the time to check its satisfiability.

We used logs with batches of one hundred events and an average of one hundred concurrent enactments, as these values were found to be within the feasible range for the multi-rule monitor in Section 4.4.3. We generated thirty sets with two to four small rules, calculating the overlap by summing the overlap for each pair of rules, then dividing by the number of rule pairs. We place each rule in one of five categories: 0.33, 0.66, 1, 1.33, and 1.66 event atoms, whichever is closest to its average overlap. In Table 4.2, we report the average batch processing time with any time above the batch arrival rate of one second shown in parentheses to indicate infeasibility. the overlap grows exponential as the overlap increases linearly. Furthermore, we see when the average overlap exceeds 1 event atom per rule pair, the processing time exceeds one second, the threshold for feasibility.

Overlap≈0.33	**Overlap≈0.66**	**Overlap≈1**	**Overlap≈1.33**	**Overlap≈1.66**
0.048	0.169	0.356	(1.22)	(3.17)

Table 4.2: Average Batch Processing Time (seconds) by Average Overlap

We also evaluated the effect of varying the number of rules while keeping a constant overlap. More rules grows the number of tables and entries in the assignment databases, as well as the number of expected events generated by the Chase, as each rule's body assignment generates expected events. The average overlap for the sets of rules was 0.54, so we used five rule sets of two, three, four, and five small rules with an overlap of 0.54 ±

0.2. We report the average batch processing times in Table 4.3. The batch processing time increases exponentially with the number of rules; this suggests the number of rules is similarly critical to the feasibility of monitoring as the overlap, where rule sets with relatively low overlap can become infeasible when the number of rules exceeds four.

2 rules	**3 rules**	**4 rules**	**5 rules**
0.014	0.099	0.395	(1.582)

Table 4.3: Average Batch Processing Time (seconds) for Overlap≈0.54 by Number of Rules

In summary, our evaluation quantifies the benefits of early violation detection and the feasibility of monitoring with respect to the size and complexity of logs and rules. We identify the algorithms and subroutines that are most affected by these dimensions and provide explanations for the observed trends. We also identify dimensions of logs and rules that deserve further investigation, such as the effects of gap size on earliness and the effects of gap size and number of gaps on processing time. Finally, we note that these findings are limited to applications with similar characteristics as the sample logs and rules we used in our evaluation, and thus may not generalize to arbitrary event-based systems. More work is needed to extend these findings to a wider range of applications, particularly by testing real-world logs and rules.

4.5 Related Work

We first discuss related work that reasons algorithmically about when a violation is inevitable. Then, we compare our work with other approaches that monitor constraints with data values.

A key technique in this work is to detect violations at the earliest possible time. References [55–57] studies violations of constraints in DECLARE language, using an encoding of violations' temporary or permanent status in states of automata. Quantitative time constraints, such as "for every request followed within five days by a response, a payment is made within three days of the response and three days of the request", are important for applications of runtime monitoring [18], but difficult to encode in automata, as evidenced by the previous chapter. We identify violations as inevitable or not by partially instantiating constraints with observed timestamps and data values and checking satisfiability of the resulting constraints. References [58] and [59] also partially initialize constraints to detect violations, but do not consider interactions between sets of constraints, which may produce earlier violations as illustrated above.

Another functionality of runtime constraints considered by this chapter is the comparison of data values, such as matching the user who makes a request to the user who receives a response. References [16, 60] monitor constraints in first-order LTL with automata whose states have relational data stores, though they assume a fixed, finite domain of data values, which is impractical for large applications. Quantified event automata, finite-state automata with transitions labeled by quantified first-order formulas, can also monitor data-dependent constraints [61]. Their approach creates and manages bindings to variables, but is limited by the same drawbacks as automata-based approaches described above. Other work on first-order LTL uses exclusively relational data structures as auxillary storage for violation detection [36,62–64], though they do not calculate deadlines explicitly because they do not use quantitative time constraints. Also relevant is the technique of trace slicing [50], filtering an enactment into disjoint enactments with related data, allowing multiple monitors to run in parallel. Our work does not use trace slicing, but it seems promising for optimization as a pre-processing step to parallelize our approach.

Other relevant data-centric approaches are those for relational databases and Datalog rules. Incremental view maintenance for Datalog provides incremental algorithms for updating the results of views or queries when the underlying database changes; [65]

maintains Datalog and SQL views without gap constraints nor existential variables and multiple atoms in the rule head. A key technique in this work comes from the observation that because we use Datalog-like rules, the possibility of rules triggering other rules can be simulated by the chase [48], which we adapt for our setting in Section 4.2. Other work on the chase for Datalog with arithmetic constraints targets problems that don't apply to our setting of monitoring an enactment at runtime, including computing certain query answers [66, 67].

4.6 Chapter Summary

This chapter presents techniques for detecting violations of individual rules and sets of rules with data, extending the class of rules that can be monitored. We showed that detecting violations at the earliest possible time. can be accomplished by reasoning about timestamps, simulating rule interaction, and applying satisfiability testing to potential violations. We also conducted an empirical evaluation of our techniques to show that they are effective and efficient for small- and medium-sized batches and rules, though more study is needed to determine their effectiveness and efficiency with enterprise-scale computing resources. In the next chapter, we explore the use of aggregation functions over time windows in rules.

Chapter 5

Rules with Aggregation

This chapter studies for rules with aggregation functions. The high volume of data and relatedness of events in enactments encourages specification that aggregate properties of groups of data. For example, a banking application may report an account is suspicious for money laundering if the sum of the account's payments in a 24-hour period exceeds \$50,000, even if no individual transaction exceeds that amount. Such rules call for monitoring techniques that use aggregation functions. These functions introduces new challenges of reasoning about functions on multiple events, as well as numeric data. In this chapter, we develop a syntax and semantics for extending our rules with aggregation functions. Then, we provide two ways of addressing the challenges of early violation detection. The first is to rewrite aggregation with Datalog$_{\mathbb{Z}}$ programs, which allows us to use the results of the previous chapter with minimal changes. The second is to adapt the chase process to reason with aggregation functions.

This chapter is organized as follows: in Section 5.1, we add time windows and aggregation functions over these windows to rule syntax and semantics. In Section 5.2, we provide Datalog$_{\mathbb{Z}}$ programs that generate events with the results of aggregation without calling the underlying aggregation functions directly. In Section 5.3, we combine the algorithms from the previous chapter to perform early violation detection, by adapting the chase process and rewriting aggregation functions in Presburger arithmetic (PA). Finally, in Sections 5.4 and 5.5 discuss related work and conclude the chapter.

5.1 Time Windows for Aggregation Functions

Now, we present the syntax for rules with aggregation, which uses different types of windows to collect events and five aggregation functions to aggregate the values of events in a window. Then, we describe the semantics of rules with aggregation, which do not just constrain event, but also generate events that hold the results of applying aggregation functions. Finally, we describe an assumption about target enactments we make to simplify the problem and a preprocessing step that enables this assumption. The results in the remainder of this chapter assume that the workflow assumption holds.

Example 5.1 : We illustrate the aggregation model with a banking application where users deposit money into their accounts. Enactments for this application include events for users' deposits and bankers' approval of users' activity, with two event types: `Deposit`, with *user* and *amount* attributes, and `Approve`, with the *user* attribute. The *user* values come from the data domain $\mathcal{D}$ and *amount* values are positive integers in $\mathbb{N}$.

Consider if the bank requires that over every three-day period: (1) the sum of deposits must be at most \$20 and (2) some deposit is more than \$10. These requirements are specified by the rules r_1 and r_2, respectively, in Figure 5.1. Rule r_1 computes the sum of all deposits for a user over each three-day window in a new "aggregation event" type `SumDep`$(sum, start)$. This event type has two attributes: *sum* is the sum of the deposits in the window and *start* is the first timestamp of the window. The gap atom $a' \leqslant 20$ in the rule head requires that the sum a' is at least \$20. Rule r_2 aggregates the maximum deposit for a user over each three-batch window in a new aggregation event type `MaxDep`$(max, start)$, where MAX is the maximum of the deposits in the window and *start* is the first timestamp of the window. The gap atom $b' \geqslant 10$ in the rule head enforces the requirement that the maximum b' is at least \$10. ∎

$$r_1 : \texttt{SumDep}(u, a' = \textsc{sum}(a), s)@(s+3),\ a' \leqslant 20 \leftarrow \textsc{tumbling}(s, s+3)$$

$$\textsc{from}\ \texttt{Deposit}(u, a)@z$$

$$r_2 : \texttt{MaxDep}(u, b' = \textsc{max}(b), s)@(s+3),\ b' \geqslant 10\ \leftarrow \textsc{tumbling}(\text{s,s+3})$$

$$\textsc{from}\ \texttt{Deposit}(u, b)@z$$

Figure 5.1: Two rules r_1 and r_2 with aggregation functions over two types of sliding windows

The rule heads contain atoms that name events that hold the results of aggregation functions and additionally may gap atoms on those events. We allow the aggregation functions sum (SUM), maximum (MAX), minimum (MIN), count (COUNT), and count-unique (COUNTU). The values $-\infty$ and ∞ are the default values for MAX and MIN, respectively; we assume these values not present in the enactment and are used when the window contains no events. The value $0 \in \mathbb{N}$ is the default value for SUM, COUNT, and COUNTU.

The *window expression*, e.g., TUMBLING$(s, s+3)$ in r_1, indicates which events are collected in windows for r_1. We use two classes of windows, either *moving* or *triggered*, which are defined by the window expression. Moving windows are either *sliding* and *tumbling* windows; they have a constant length and are evaluated at every timestamp or at every timestamp that is a multiple of the window length, respectively, regardless of the enactment's events. The *sliding* window is parametrized by its window length L for some $L \geqslant 1$; it generates a window for the interval of timestamps $[1, L]$, and then, for each integer i, the interval i through $i + L$, assuming the enactment extends to $i + L$. The *tumbling* window is defined by a window length L, and includes the window $[1, L]$, then the window $[kL+1, kL+L]$ for every integer $k \geqslant 1$. Fig. 5.2 shows an example of moving windows. On the top of the figure, `Deposit` source events appear at each timestamp. Sliding and tumbling windows are shown as boxes of length 3, with the window's aggregation event inside the box.

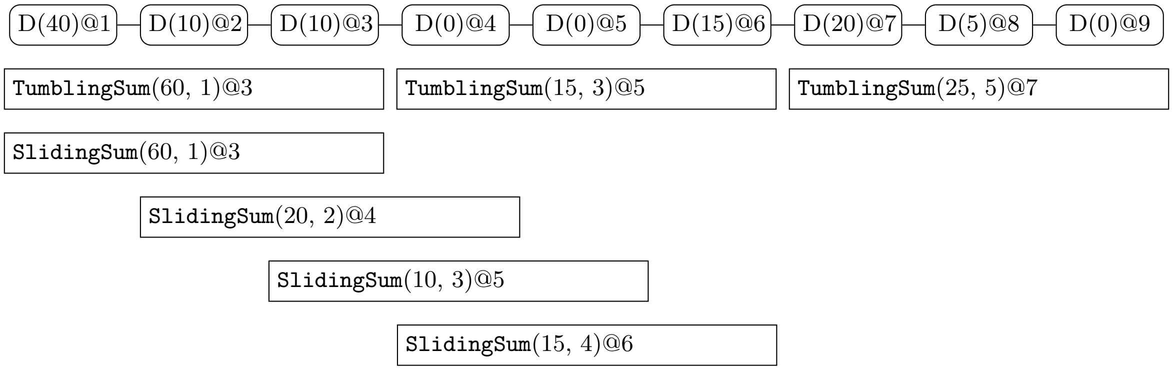

Figure 5.2: An enacment with `Deposit` (`D`) events, tumbling and sliding windows for window length 3, and aggregation events `TumblingSum` and `SlidingSum`

The second class of windows are those with event triggers. For a *start-triggered* window of length L, a window $[s, s+L]$ is generated whenever a specified event occurs at time s. Similarly, for an *end-triggered* window, a window $[e-L, e]$ is generated whenever a specified event occurs at time e. Finally, a *start-end-triggered* window indicates a window between every pair of start and end events. Consider the enactment and corresponding windows in Figure 5.3. The start trigger `B` and the end trigger `C` events are shown above the `Deposit` source events. Windows are shown as boxes, with vertical, dashed Lines matching each window to its start and end events.

The rule semantics for aggregation expressions are different from the semantics for non-aggregation expressions: rules with aggregation generate "aggregation events". To define these semantics, we refine the event model to distinguish between *external* or *internal* events. External events are generated by an outside source; this is the event model in the previous chapters. Internal events are generated by the monitoring system, here for each window of an aggregation rule. For an enactment η and an aggregation

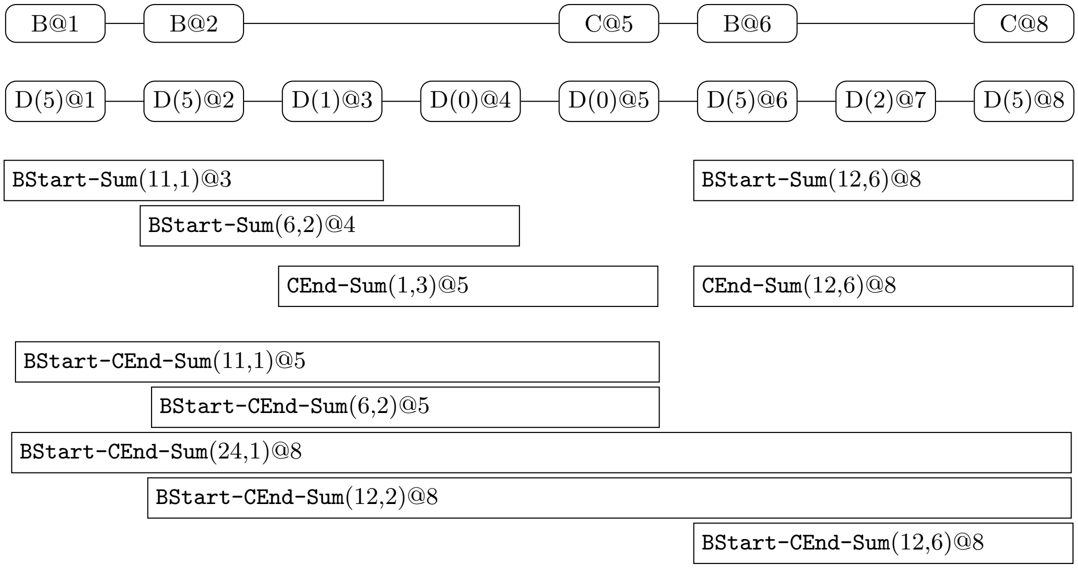

Figure 5.3: Triggered windows, triggered by start event `B` and end event `C`

rule r, for each window W defined by the window expression in $\mathit{body}(r)$, an internal *aggregation event* is generated with the timestamp of the last event in W and the value of the aggregation function in $\mathit{head}(r)$ applied to data values for the attribute and source events in W designated by the rule body.

This new presence of aggreagtion functions require reasoning about collections of events, including overlapping windows. Furthermore, the results of aggregation functions are numeric values; previously we only needed to reason about data values in $\mathcal{D}$ and timestamps. This motivates the development of extension of the chase process and satisfiability checking to rules with aggregation functions. Thus, the technical problem addressed by this chapter is: given a set of rules with aggregation functions, report an enactment's violations at the earliest possible time.

As a simplifying assumption, we assume that for each event type that acts as a source for an aggregation rule, there is exactly one event of that type per timestamp. We call this the *workflow assumption*, also referred to as the DECLARE assumption in [68] and the *simplicity* assumption in [69]. This assumption limits the types of enactments to which our results apply, but it is needed for the correctness of the Datalog$_{\mathbb{Z}}$ programs in the algorithms in the following section. Additionally, we use a preprocessing step that modifies an input enactment with *at most one* event per type per timestamp to produce an enactment where each event type has *exactly one* event per timestamp. This preprocessing step adds an attribute *real* to each event type. If an event of that type at a timestamp is present in the input enactment, *real* is set to 1. Otherwise, a placeholder event is added with the value 0 for *real* and 0 for all other attributes. We give an example of this preprocessing in Fig. 5.4. Our results in the remainder of this chapter assume that the workflow assumption holds.

timestamp	Source events	Preprocessed events
1	$\texttt{Deposit}(10)@1$	$\texttt{Deposit}(10, 1)@1$
2	$\texttt{Deposit}(40)@2$	$\texttt{Deposit}(40, 1)@2$
3		$\texttt{Deposit}(0, 0)@3$
4	$\texttt{Deposit}(10)@4$	$\texttt{Deposit}(10, 1)@4$
5		$\texttt{Deposit}(0, 0)@5$

Figure 5.4: Preprocessing by adding an attribute *real* to each event type.

5.2 Datalog$_{\mathbb{Z}}$ Generation of Aggregation Events

In this section, we present Datalog$_{\mathbb{Z}}$ programs [70] to generate aggregation events. This variant of Datalog allows integer constants and arithmetic expressions in the head and body of rules. For consistency with our language, we use the same "@" syntax for the event's timestamps. We provide programs for MAX for sliding, tumbling, start-triggered, and end-triggered windows; COUNT for sliding windows, and SUM for sliding windows. The other functions and window types are handled with similar programs. This provides a means of early violation detection for rules with aggregation functions, by rewriting rules with aggregation as Datalog$_{\mathbb{Z}}$ programs and using these programs to generate aggregation events, then applying the algorithms in the previous chapter without modification.

The structure of the Datalog$_{\mathbb{Z}}$ program is consistent across the different types of windows: some event `incX` with an "accumulator" attribute stores the partial result of the aggregation function over the partial window, up to the current timestamp. To compute `incX`, some rules initialize `incX` with the first source event's value, then other rules propagate `incX` with `incX` from the previous timestamp and the current source event's value. Finally, a rule reports `resultX` when the window is complete. In these programs, we simplify the presentation by removing the non-aggregated attributes from the source (`Src`) events, and use `V` for the target attribute.

5.2.1 Sliding window with MAX function

Let `Src` be the source event and $L + 1$ the sliding window size for a rule with the MAX function, i.e., the rule has the form of Fig. 5.5.

$$\texttt{resultSlidingMax}(\textsc{max}(a), s)@(s+L) \leftarrow \textsc{Over sliding}(s, s+L)$$
$$\textsc{From}\ \texttt{Src}(a)@x$$

Figure 5.5: Rule with the MAX function and a sliding window of size $L + 1$.

An internal event `resultSlidingMax` is generated for each window, with attributes *value* (the maximum value), *start* and *end* (of the window), and *time* (when the result is produced) for each window [*start*, *end*] of length $L + 1$. To compute `resultSlidingMax` incrementally, we use an internal event `incSlidingMax` with attributes *accumulator*, *start*, *end*, and *stop*, where *accumulator* holds the maximum value seen from *start* to *stop*, and *end* attribute is the target window's last timestamp, i.e., when the result should be reported. The Datalog$_{\mathbb{Z}}$ program (Fig. 5.6) initializes `incSlidingMax` at each timestamp T with the start of a sliding window. The source event's value is stored in the accumulator (of $-\infty$ if the source event is not real), and $T{+}L$ as the window end. At each timestamp T that is not greater than the window end, if the source event at T is real, the accumulator's value at T is compared with the source event's value at $T{+}1$. Otherwise, the accumulator is passed along. At the window's end, the accumulator is reported with `resultSlidingMax`.

```
Initialize incSlidingMax
incSlidingMax(V, T, T+L)@T     ← Src(V, 1)@T
incSlidingMax(−∞, T, T+L)@T    ← Src(V, 0)@T

Propagate incSlidingMax
incSlidingMax(V, S, E)@(T+1)   ← incSlidingMax(A, S, E)@T,  Src(V, 1)@(T+1),  (A<V),  (T+1⩽E)
incSlidingMax(A, S, E)@(T+1)   ← incSlidingMax(A, S, E)@T,  Src(V, 1)@(T+1),  (A⩾V),  (T+1⩽E)
incSlidingMax(A, S, E)@(T+1)   ← incSlidingMax(A, S, E)@T,  Src(V, 0)@(T+1),  (T+1⩽E)

Report resultSlidingMax
resultSlidingMax(A, S)@E       ← incSlidingMax(A, S, E)@T,  (T=E)
```

Figure 5.6: Datalog$_{\mathbb{Z}}$ program for MAX and a sliding window $[S, E]$ of size L

The results of evaluating the Datalog$_{\mathbb{Z}}$ program in Fig. 5.7 with a window size 5 is shown in Fig. 5.7. The source events are shown in the first column. The second column shows the `incSlidingMax` events for the first window $[1, 5]$. The third column shows the `incSlidingMax` events for the second window $[2, 6]$. The fourth column shows the `resultSlidingMax` event for the first window $[1, 5]$, which appears when the source event at timestamp 5 is processed, as well as for the second window $[2, 6]$.

time	Source events	`incSlidingMax` events		`resultSlidingMax` events
1	`Src`(0,0)@1	`incSlidingMax`$(-\infty, 1)$@1		
2	`Src`(5,1)@2	`incSlidingMax`$(5, 1, 5)$@2	`incSlidingMax`$(5, 2, 6)$@2	
3	`Src`(0,0)@3	`incSlidingMax`$(5, 1, 5)$@3	`incSlidingMax`$(5, 2, 6)$@3	
4	`Src`(0,0)@4	`incSlidingMax`$(5, 1, 5)$@4	`incSlidingMax`$(5, 2, 6)$@4	
5	`Src`(15,1)@5	`incSlidingMax`$(15, 1, 5)$@5	`incSlidingMax`$(15, 2, 6)$@5	`resultSlidingMax`$(15, 1)$@5
6	`Src`(0,0)@6		`incSlidingMax`$(15, 2, 6)$@6	`resultSlidingMax`$(15, 2)$@6

Figure 5.7: Evaluating the sliding, MAX program for window $[S, E]$ of size 5

5.2.2 Tumbling window with MAX function

Let Src be the source event and $L+1$ the tumbling window size, i.e., the rule has the form of Fig. 5.8.

$$\begin{array}{rl}\texttt{resultTumblingMax}(\textsc{max}(a), s)@(s+L) \leftarrow & \textsc{Over tumbling}(s, s+L) \\ & \textsc{From } \texttt{Src}(a)@x\end{array}$$

Figure 5.8: Rule for tumbling window with size L+1 and MAX function

To compute the corresponding internal event resultTumblingMax incrementally, we use an internal event incTumblingMax with the same attributes as incSlidingMax. We use a Datalog$_{\mathbb{Z}}$ program (Fig. 5.9) to generate incTumblingMax events, which only differs from that for the sliding window in that when an window is initialized, the next window is initialized starting L timestamps later rather than at the next timestamp.

Initialize incTumblingMax

$$\begin{array}{ll}
\texttt{incTumblingMax}(V, 0, L)@0 & \leftarrow \texttt{Src}(V, 1)@0 \\
\texttt{incTumblingMax}(-\infty, 0, L)@0 & \leftarrow \texttt{Src}(V, 0)@0 \\
\texttt{incTumblingMax}(V, T{+}1, T{+}1{+}L)@T & \leftarrow \texttt{incTumblingMax}(A, S, E)@T,\ (E{=}T), \texttt{Src}(V, 1)@(T{+}1) \\
\texttt{incTumblingMax}(-\infty, T{+}1, T{+}1{+}L)@T & \leftarrow \texttt{incTumblingMax}(A, S, E)@T,\ (E{=}T),\ \texttt{Src}(V, 0)@(T{+}1)
\end{array}$$

Propagate incTumblingMax

$$\begin{array}{ll}
\texttt{incTumblingMax}(V, S, E)@(T{+}1) & \leftarrow \texttt{incTumblingMax}(A, S, E)@T, \texttt{Src}(V)@(T{+}1), (A{<}V), (T{+}1{\leqslant}E) \\
\texttt{incTumblingMax}(A, S, E)@(T{+}1) & \leftarrow \texttt{incTumblingMax}(A, S, E)@T, \texttt{Src}(V)@(T{+}1), (A{\geqslant}V), (T{+}1{\leqslant}E)
\end{array}$$

Report resultTumblingMax

$$\begin{array}{ll}
\texttt{resultTumblingMax}(A, S)@E & \leftarrow \texttt{incTumblingMax}(A, S, E)@T,\ (T{=}E)
\end{array}$$

Figure 5.9: Datalog$_{\mathbb{Z}}$ program for MAX function on tumbling window $[S, E]$ of size L

5.2.3 Start-and end-triggered window for MAX function

For a start- and end-triggered window $[s, e]$ with the MAX function, let Src be the source event, Start the start-trigger event, and End the end-trigger event, i.e., the rule has the form of Fig. 5.10.

$$\begin{array}{l} \texttt{resultStartEndMax}(\textsc{max}(a), s)@e \leftarrow \textsc{Over}(\texttt{Start}@s, \texttt{End}@e) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textsc{From}\ \texttt{Src}(a)@x \end{array}$$

Figure 5.10: Rule with the MAX function, start- and end-triggered window $[s, e]$

We use a Datalog$_{\mathbb{Z}}$ program (Fig. 5.11) to generate `incStartEndMax` events as follows: Whenever a start trigger event is observed, an `incStartEndMax` event is initialized using the start trigger event's timestamp T as the window start, placing the source event's value in the *accumulator*. Data in `incStartEndMax` at a timestamp T is propagated to the next timestamp $T{+}1$ using the source event at timestamp $T{+}1$. To do this, if the source event is real, the accumulator is compared with the value of the source event at timestamp $T{+}1$ and the larger is passed along. If the source event is not real, the accumulator is passed along. We report `resultStartEndMax` when END event arrives using the larger of the accumulator from the previous timestamp's `incStartEndMax` and the source event's value.

$$\begin{array}{ll}
\text{Initialize } \texttt{incStartEndMax} & \\
\texttt{incStartEndMax}(V, T)@T & \leftarrow \texttt{Src}(V, 1)@T,\ \texttt{Start}@T \\
\texttt{incStartEndMax}(-\infty, T)@T & \leftarrow \texttt{Src}(V, 0)@T,\ \texttt{Start}@T \\
& \\
\text{Propagate } \texttt{incStartEndMax} & \\
\texttt{incStartEndMax}(V, S)@(T{+}1) & \leftarrow \texttt{incStartEndMax}(A, S)@T,\ \texttt{Src}(V, 1)@(T{+}1),\ (A{<}V) \\
\texttt{incStartEndMax}(A, S)@(T{+}1) & \leftarrow \texttt{incStartEndMax}(A, S)@T,\ \texttt{Src}(V, 1)@(T{+}1),\ (A{\geqslant}V) \\
\texttt{incStartEndMax}(A, S)@(T{+}1) & \leftarrow \texttt{incStartEndMax}(A, S)@T,\ \texttt{Src}(V, 0)@(T{+}1) \\
& \\
\text{Report } \texttt{resultStartEndMax} & \\
\texttt{resultStartEndMax}(A, S)@E & \leftarrow \texttt{incStartEndMax}(A, S)@T,\ \texttt{End}@T
\end{array}$$

Figure 5.11: Datalog$_{\mathbb{Z}}$ program for `incStartEndMax` and `resultStartEndMax` events

5.2.4 Sliding window with SUM function

Let `Src` be the source event and $L+1$ the window size for a sliding rule with the SUM function, i.e., the rule has the form of Fig. 5.12.

$$\texttt{resultSlidingSum}(\textsc{sum}(a), s)@(s+L) \leftarrow \textsc{Over sliding}(s, s+L)$$
$$\textsc{From } \texttt{Src}(a)@x$$

Figure 5.12: Rule for sliding window with SUM function

To compute `resultSlidingSum` incrementally, we use an internal event `incSlidingSum` with attributes *accumulator*, *start*, *end*, and *stop*. We use a Datalog$_{\mathbb{Z}}$ program (Fig. 5.13) to generate `incSlidingSum` events as follows: each timestamp is the start of a sliding window so for each source event at timestamp T, an `incSlidingSum` event is initialized using a source event's timestamp as the window start, the source event's value in the accumulator, and T+L as the window end. Data in `incSlidingSum` at a timestamp T is propagated to the next timestamp T+1 using the source event at timestamp T+1. To do this, the accumulator is added with the value of the source event at timestamp T+1. The propagation only happens if the next timestamp is no greater than the window end. When the source timestamp is equal to the window end, the value in the accumulator is reported with `resultSlidingSum`.

5.2.5 The Datalog$_{\mathbb{Z}}$ Programs Compute Aggregation Events

Given the above Datalog$_{\mathbb{Z}}$ programs, we argue they compute the aggregation events for their respective rules. This is done by showing that the program's incremental computations correctly compute the corresponding aggregation function over the appropriate window and that the result is reported at the correct time.

Theorem 5.1 : For each aggregation rule r, there is a Datalog$_{\mathbb{Z}}$ program that generates the aggregation events for r.

```
Initialize incSlidingSum
incSlidingSum(V, T, T+L)@T         ← Src(V, 1)@T
incSlidingSum(0, T, T+L)@T         ← Src(V, 0)@T

Propagate incSlidingSum
incSlidingSum(A+V, S, E)@(T+1)  ← incSlidingSum(A, S, E)@T,  Src(V, 1)@(T+1),  (T+1⩽E)
incSlidingSum(A, S, E)@(T+1)      ← incSlidingSum(A, S, E)@T,  Src(V, 0)@(T+1),  (T+1⩽E)

Report resultSlidingSum
resultSlidingSum(A, S)@E           ← incSlidingSum(A, S, E)@T,  (T = E)
```

Figure 5.13: Datalog$_{\mathbb{Z}}$ Program `incSlidingSum` and `resultSlidingSum`

Proof: We use the above programs as templates and show that each program generates the aggregation events for a window of size L. We do this by induction on the value of l.

Base case: $L = 1$, i.e., the window is a single timestamp. For the MAX function, observe that `incSlidingMax` is initialized with the source event's value (or a negative infinity if the source event is not present), for the window with start T and end T. Because the window is a single timestamp, the end timestamp is equal to the start timestamp. During propagation, the accumulator is kept the same if the source event is not present, i.e., $\texttt{Src}(*, 0)$. If the source event is real, i.e., $\texttt{Src}(*, 1)$, the accumulator is changed to the source event's value if and only if the source event's value is greater than the accumulator, e.g., $A<V$, otherwise the accumulator is kept the same. A similar argument holds for MIN.

For COUNT, `incSlidingCount` is initialized with 1 if the source event is present. Otherwise, it is initialized with 0. In the propagate rules, the accumulator is updated with $A + 1$ if and only if the source event is present. Otherwise, the accumulator is kept the same. A similar argument holds for SUM and COUNTU.

Inductive step: $L > 1$. For MAX, we assume that the program generates the aggregation events for a window of size $L{-}1$. This is only possible if `incSlidingMax` holds the maximum value of the first $L-1$ source events of the window. The propagation rule compares the latest source event in in the window with the value in the accumulator. If the source event is greater than the accumulator, then the accumulator is updated with the source event's value. Otherwise, the accumulator is not updated. A similar argument holds for MIN.

For COUNT, we assume that the program generates the aggregation events for a window of size $L-1$. This is only possible if `incSlidingCount` holds the count of the first $L-1$ source events of the window. The propagation rule increments the accumulator if the latest source event in the window is not a placeholder. Otherwise, the accumulator keeps the same value. A similar argument holds for SUM and COUNTU. ▮

5.3 Chasing Rules with Aggregation

In this section, we describe how to apply the chase directly to rules with aggregation. We present techniques to extend the algorithms of the previous sections with aggregation functions, which requires reasoning about arithmetic functions applied to a numeric domain and time windows that include future events. First, we describe when to generate "expected" source events to fill an open time window. Then, we rewrite aggregation functions as Presburger arithmetic (PA) constraints on a given window of source events. Finally, we include these constraints in the satisfiability test that detects violations. These techniques, along with the algorithms from the previous chapter, is sufficient for early violation detection of acyclic sets of rules with aggregation.

5.3.1 Assignments and Expected Events for Rules with Aggregation

The fundamental problem of violation detection remains to match body assignments with head assignments, though assignment creation may now use values derived from aggregation functions applied to a window of source events. We illustrate this with a running example, using the same assignment data structures $\textsc{ba}_r$, $\textsc{ha}_r$, and $\textsc{ext}_r$ for each rule r from the previous chapter.

timestamp	Alice's events	Bob's events
1	`Deposit`(Alice, 9)@1	`Deposit`(Bob, 3)@1
2	`Deposit`(Alice, 6)@2 `Approve`(Alice, 20)@2	`Deposit`(Bob, 5)@2

Figure 5.14: An example enactment η for two users

$$
\begin{aligned}
r_1 : \texttt{SumDep}(u, a' = \textsc{sum}(a), s)@(s+3),\ a' \leqslant 20 &\leftarrow \textsc{tumbling}(s, s+3) \\
&\qquad \textsc{from}\ \texttt{Deposit}(u, a)@z \\
r_2 : \texttt{MaxDep}(u, b' = \textsc{max}(b), s)@(s+3),\ b' \geqslant 10 &\leftarrow \textsc{tumbling}(\mathrm{s,s+3}) \\
&\qquad \textsc{from}\ \texttt{Deposit}(u, b)@z \\
r_3 : \text{Approve}(u, c)@(x-1) &\leftarrow \mathit{SumDep}(u, c)@x, c \geqslant 18
\end{aligned}
$$

Figure 5.15: Three rules, two with aggregation

Example 5.2 : Consider the enactment η in Figure 5.14. and the rules in Figure 5.15. Rule r_1 aggregates the total amount of deposits per three-day period and requires this total to be under \$20. Rule r_2 aggregates the maximum deposit amount per three-day period and require this maximum to be at least \$10. Rule r_3 requires that users' three-day deposit totals are approved when the total is at least \$18.

For these rules, Update creates assignments for variables u, a', s, a, and z in r_1, to v, b', t, b, and z in r_2, and to u, c, and x in r_3, by matching event instances with event atoms. From the event `Deposit`(Alice, 9)@1 and the body event atom $\texttt{Deposit}(u, a)@z$, the partial assignment $\alpha_1 = \{u \mapsto \text{Alice}, a \mapsto 9, z \mapsto 1\}$ is created, which is Row 1 of the table in Fig. 5.16 From event `Deposit`(Alice, 6)@2 and the same event atom, we create the partial assignment $\alpha_2 = \{u \mapsto \text{Alice}, a \mapsto 6, z \mapsto 2\}$ (Row 2). Fig. 5.16 shows some assignments in $\textsc{ba}_{r_0}(\eta)$ at time 2. ∎

$\textsc{ba}_{r_1}(\eta)$

id	u	a	z	constraints	match?	chased?
μ_1	Alice	9	1	-	*no*	*no*
μ_2	Alice	6	2	-	*no*	*no*
μ_3	Bob	3	1	-	*no*	*no*
μ_4	Bob	5	2	-	*no*	*no*

Figure 5.16: The $\textsc{ba}_r$ table for r_1 and η at time 2

Now, we describe how "expected" events are added to the enactment through a chase process. This is similar to the chase in the previous chapter, but in this setting, more expected events may be created, one event for each future source event in the corresponding window. That is, when the window is "open" with respect to the current time, i.e., its start timestamp is in the past or present and its end timestamp is in the future, one expected source event (with marked nulls) are added to the enactment for each of the open window's future timestamps. Then, the relevant aggregation function is applied to the source events' values, creating an aggregation event. The expected source events have marked nulls for their data values, so the aggregation function may be applied to both known values and marked nulls. Example 5.3 illustrates extending a window with expected source events, then generating an aggregation event with a rule.

Example 5.3 : Continuing with the enactment in Example 5.2, at time 2, the window $(1, 4)$ is open for aggregation rules r_1 and r_2. We add expected `Deposit` events for Alice and Bob at time 3 with null values a_3 and b_3, resp., this is shown in Fig. 5.17. Given values for all source events in the window $(1, 4)$ for r_1 and r_2, r_1 yields the body

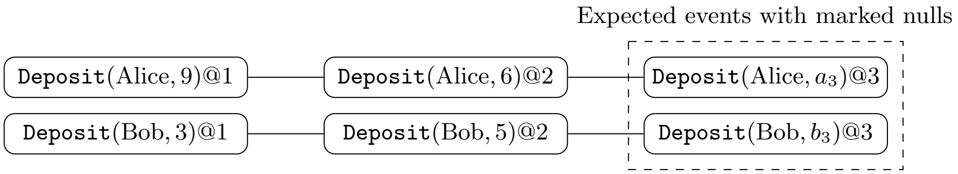

Figure 5.17: Adding expected source events to an open window $(1, 3)$

assignment α_a in Fig. 5.18. This produces the head assignment $\beta_a = \{s \mapsto 1, u \mapsto Alice, a' \mapsto a'_a\}$ with the constraints $a'_a = \textsc{sum}(9, 6, a_3)$ and $a'_a \leqslant 20$ in Fig. 5.19. and the expected event $\texttt{SumDep}(Alice, a'_a)$@3. Rewriting $\textsc{sum}$ in PA, we have the constraint $(a'_a = 9 + 6 + a_3) \wedge (a'_a \leqslant 20)$. For Bob, we have similar body assignment α_a and head assignment β_a.

id	u	s	constraints	match?	chased?
α_a	Alice	1	-	no	no
α_b	Bob	1	-	no	no

Figure 5.18: Complete assignments in BA for r_1 and η at time 2

id	u	a'	s	constraints
β_a	Alice	a'_a	1	$(a'_a = 9 + 6 + 3) \wedge (a'_a \leqslant 20)$
β_b	Bob	a'_b	1	$(a'_b = 6 + 5 + 5) \wedge (a'_b \leqslant 20)$

Figure 5.19: Complete assignments in HA for r_1 and η at time 2

For the body assignment γ_a for r_2, we get the head assignment $\delta_a = \{s \mapsto 1, u \mapsto Alice, b' \mapsto b'_a\}$ and the expected event $\texttt{MaxDep}(Alice, b'_a)$@3 Rewriting $\textsc{max}$ in PA, we obtain that δ_a has constraint $((b'_a \geqslant 9) \wedge (b'_a \geqslant 6) \wedge (b'_a \geqslant a_3)) \wedge ((b'_a = 9) \vee (b'_a = 6) \vee (b'_a = a_3)) \wedge (b'_a \geqslant 10)$. For Bob, we have similar head assignments γ_b and δ_b. This leads to the $\textsc{ba}_{r_2}(\eta)$ and $\textsc{ha}_{r_2}(\eta)$ entries in Figures 5.20 and 5.21 at time 2. ▌

5.3.2 Rewriting Aggregation Atoms

When an open window includes expected source events, the aggregation function's result depends their unknown values. These unknown values are represented by marked nulls, and even if their exact values are unknown, the marked nulls may be constrained by the rule's gap atoms or the chase process, leading to violations. To reason with

id	u	s	constraints	match?	chased?
γ_a	Alice	1	-	no	no
γ_b	Bob	1	-	no	no

Figure 5.20: Complete assignments in BA for r_2 and η at time 2

id	u	b'	s	constraints
δ_a	Alice	b'_a	1	$((b'_a \geqslant 9) \wedge (b'_a \geqslant 6) \wedge (b'_a \geqslant a_3)) \wedge ((b'_a = 9) \vee (b'_b = 6) \vee (b'_a = a_3)) \wedge (b'_a \geqslant 10)$
δ_b	Bob	b'_b	1	$((b'_b \geqslant 3) \wedge (b'_b \geqslant 5) \wedge (b'_b \geqslant b_3)) \wedge ((b'_b = 3) \vee (b'_b = 5) \vee (b'_b = b_3)) \wedge (b'_b \geqslant 10)$

Figure 5.21: Complete assignments in HA_r table for r_2 and η at time 2

these constraints, we rewrite each aggregation atom over a given window in Presburger arithmetic in the chase step. We illustate this rewriting, then state that this rewriting preserves the semantics of the aggregation function in Lemma 5.2.

Example 5.4 : Continuing the running example, after introducing the expected event `Deposit`(Alice)(a_3)@3, the corresponding body assignment is chased to produce the aggregation event `SumDep`(Alice)(a'_a)@3 with constraint $(a'_a = 9 + 6 + a_3) \wedge (a'_a \leqslant 20)$. This generates the complete assignment $\omega_1 = \{u \mapsto \text{Alice}, x \mapsto 3\}$ for the body of r_3, where a_3 in the `Deposit` event may be 12. We chase r_3 with ω_3, propagating the assumption that $a_3 = 12$, to produce the head assignment $\tau_1 = \{u \mapsto \text{Alice}, x \mapsto 3\}$ and an expected event `Approve`(Alice, c_a)@3, along with constraints $(a'_a = 9 + 6 + a_3) \wedge (a'_a \leqslant 20) \wedge (a_3 = 12) \wedge (c_a = a'_a)$.

For Bob's events, the same process for a complete assignment ω_2 produces the head assignment $\tau_2 = \{u \mapsto \text{Bob}, x \mapsto 3\}$ and an expected event `Approve`(Bob, c_b)@3, along with constraints $(b'_b = 3 + 5 + b_3) \wedge (b'_b \leqslant 20) \wedge (b_3 = 12) \wedge (c_b = a'_b)$. ▮

id	u	s	constraints	matched	chased
ω_a	Alice	3	$c_a = a'_a$	no	no
ω_b	Bob	3	$c_b = a'_b$	no	no

Figure 5.22: Complete assignment in BA_r table for r_3 and η at time 2

Now we describe the rewriting of aggregation atoms. We assume that the aggregation atom a has a window W of L source events. We define a function $\mathsf{Rewrite}(W, a)$ (Fig 5.24)

id	u	c	s	constraints
τ_a	Alice	c_a	1	$(a'_a = 9 + 6 + a_3) \wedge (a'_a \leqslant 20) \wedge (c_a = a'_a)$
τ_b	Bob	c_b	1	$(a'_b = 3 + 5 + b_3) \wedge (a'_b \leqslant 20) \wedge (c_b = a'_b)$

Figure 5.23: Complete assignment in $\textsc{ha}_r$ table for r_3 and η at time 2

that rewrites a in PA. Lemma 5.2 states that $\mathsf{Rewrite}(W, a)$ is equivalent to a with respect to the window W.

Lemma 5.2 : Let $W = \{\mathtt{Src}(a_1, r_1)@t_1, \mathtt{Src}(a_2, t_2)@t_2, \ldots, \mathtt{Src}(a_L, r_L)@t_L\}$ be a window of L source events, where a_i is the value of the aggregation attribute in the i-th source event and r_i is the value of the *real* attribute in the i-th source event. Let a be an aggregation atom with a window W. Then, the aggregation function on W yielding a is equivalent to $\mathsf{Rewrite}(a, W)$.

$$
\begin{array}{ll}
\textsc{fun}(a, w) & \quad \mathsf{Rewrite}(a,\ W) \\
\textsc{sum}(a, W) & \equiv (a = a_1 + a_2 + \cdots + a_L) \\
\textsc{max}(b, W) & \equiv (\bigvee_{1\leqslant i\leqslant L} a_i = b) \wedge (\bigwedge_{1\leqslant i\leqslant L} a_i \leqslant a) \\
\textsc{min}(c, W) & \equiv (\bigvee_{1\leqslant i\leqslant L} a_i = c) \wedge (\bigwedge_{1\leqslant i\leqslant L} a_i \geqslant c) \\
\textsc{count}(d, W) & \equiv (0 \leqslant d \leqslant L) \wedge \bigvee_{A\subseteq\{1,\ldots,L\},|A|=d}((\bigwedge_{j\in A} r_j{\neq}0) \wedge (\bigwedge_{j\notin A,1\leqslant j\leqslant L} r_j{=}0)) \\
\textsc{countu}(e, s, e) & \equiv (0 \leqslant u \leqslant L) \wedge \bigvee_{A\subseteq\{1,\ldots,L\},|A|=u}(\bigwedge_{j\in A}(r_j{\neq}0 \wedge \bigwedge_{k\in A,k\neq j}(a_j{\neq}\ a_k)) \\
 & \qquad\qquad \wedge(\bigwedge_{j\in\{1,\ldots,L\}-A}(r_j{=}0) \vee \exists\ l \in A.a_j = a_l))
\end{array}
$$

Figure 5.24: $\mathsf{Rewrite}$ for each aggregation function using Presburger arithmetic

Proof: In Fig 5.24, the $\textsc{sum}$ function is rewritten as the sum of the source event's values. The $\textsc{max}$ and $\textsc{min}$ functions are rewritten as a disjunction of the source event's values, requiring that the maximum or minimum value is (1) greater than or equal to (resp., less than or equal to) all source values and (2) equal to some source value. The $\textsc{count}$ function creates a disjunction over all subsets A of size d of the source events, requiring that A's source values are not placeholders and all other source values are placeholders, i.e., their *real* values are equal to zero; the disjunction is true when d is equal to the number of source events in A. The COUNTU function operates in the same manner as COUNT, but requires that the source values for non-placeholder events in A are unique. ∎

Now we describe how the rewriting of aggregation rules is used along with the extension of open windows to rewrite aggregation atoms during the chase as Chase-Agg (Algorithm 7). The algorithm for aggregation rules differs from that for non-aggregation rules by Lines 4 through 10. Line 4 identifies when a window defined by an aggregation rule is open and incomplete, meaning not all of its source events have been received. Line 5 adds future source events for the window to the set of expected events with fresh, marked nulls for data values. Lines 6, 7, and 8 compute a complete head assignment from these events for any applicable rule and window, rewriting the aggregation function in the rule head as a set of PA constraints. Line 9 adds the aggregation event to the set of expected events. This new assignment and the aggregation events are then used to update the assignments in the assigment database.

The chase may not terminate for some sets of rules and enactments, as marked nulls may create more marked nulls when rules are applied, and this may continue indefinitely. This poses a problem for using the chase as a subroutine in a violation detection algorithm. To ensure the chase terminates, we place a sufficient condition on the rules such that a marked null created by chasing the rules cannot create another marked null for the same attribute and event type. This restriction is a sufficient condition for termination of the chase for tuple-generating dependencies (Theorem 3.9 in [49]). Accordingly, we consider only acyclic sets of rules with aggregation, with the following definition of acyclic, adapted from [49], that extends the definition of acyclic used in Chapter 4.

Let R be a set of rules: the graph $G_R = (V, E)$ is defined as follows.

- V is a set of vertices (P, a) where P is an event name and a is an attribute of

Algorithm 7 Chase-Agg(Δ, $D_R(\eta)$)

```
Input: A batch $\Delta$ of events for $\eta$, rules $R$,
       the assignment database for $R$
Output: the chased assignment database $D_R\eta$ for $\eta$

if no $\textsc{ba}_r(\eta)$ contains no unmatched, complete body assignments for any $r$ then return $D_R(\eta)$
end if
while the chase is not finished do
    Let $ExpectedEvents := \varnothing$
    if $\mathsf{ts}_\Delta$ is in $W$ for some window $W$ for $r \in R$ and $W$ is not complete then
        Add the source events that complete $W$ to $ExpectedEvents$
        for each aggregation rule $r$ do
            for each body assignment $\mu$ corresponding to window $W$ of $r$ do
                Create an assignment $\mu'$ for $head(r)$ with the window with constraints $\textsc{Rewrite}(W, a)$
                Add all atoms in $\mu'(head(r))$ to $ExpectedEvents$
            end for
        end for
    end if
    for each complete, unchased body assignment $\mu \in \textsc{ba}_r(\eta)$ with no ground matching head assignment for a
  rule $r$ do
        if $r$ is not an aggregation rule then
            Create an assignment $\mu'$ for $head(r)$ with fresh, marked nulls for each existential variable
        end if
        Instantiate $head(r)$ with $\mu'$
        Add all atoms to $ExpectedEvents$
        Change the $Chased$ column of $\mu$ to "yes"
    end for
    if $ExpectedEvents$ is empty then
        exit the while loop
    end if
    Update all $\textsc{ba}_r(\eta), \textsc{ha}_r(\eta)$ tables using Update and $ExpectedEvents$ as the batch of new events
    Update all $\textsc{ext}_r(\eta)$ tables using Update-E and $ExpectedEvents$ as the batch of new events
end while
return $D_R(\eta)$
```

P. We call each (P, a) a position. For an aggregation atom P with a term, e.g., $u' = \textsc{sum}(u)$, we add the vertex (P, u) and the vertex (P, u') to V.

- E is a set of edges where for every rule $\psi(\bar{x}, \bar{y}) \leftarrow \phi(\bar{x})$ in R, we call each x in $\bar{x}$ a propagated variable. For each propagated variable x, for each occurrence of x in $\phi(\bar{x})$ in position (P, a), do two things:

  1. for each occurrence of x in $\psi(\bar{x}, \bar{y})$ at position (Q, b), add an edge from (P, a) to (Q, b), and

  2. for each existentially quantified variable y in $\psi(\bar{x}, \bar{y})$, for each occurrence of y in $\psi(\bar{x}, \bar{y})$ at position (S, c), add a *special edge* from (P, a) to (S, c).

We say R is *acyclic* if G_R has no cycle containing a special edge. Note again that this is a different definition of acyclic than the one used in Chapter 3, but an generalization of the definition used in Chapter 4.

5.3.3 Detecting Violations via Satisfiability Testing

The assignment database for an enactment can determine if an acyclic set of rules is violated using the Build algorithm from the previous chapter (Algorithm 6). The algorithm creates a PA constraint that is satisfiable if and only if the enactment has a violation. We demonstrate how this satisfiability test works in the presence of PA formula for the running example.

Example 5.5 : Applying Build to the assignment database for Alice at time 2 uses assignments α_a for r_1 and γ_a for r_2. a constraint Θ is initialized to *true*. Then, from α_a and its match β_a, $true \rightarrow (a'_a = 9 + 6 + a_3) \wedge (a'_a \leqslant 20)$ is added to Θ. From γ_a and its match δ_a, $true \rightarrow ((b'_a \geqslant 9) \wedge (b'_a \geqslant 6) \wedge (b'_a \geqslant a_3)) \wedge ((b'_a = 9) \vee (b'_b = 6) \vee (b'_a = a_3)) \wedge (b'_a \geqslant 10)$. From ω_a, there are no complete, ground, matching head assignments, so we add $\neg((a'_a = 9 + 6 + a_3) \wedge (a'_a \leqslant 20) \wedge (c_a = 12))$.

This formula Θ is not satisfiable; Θ contains $9 + 6 + a_3 = a'_a \leqslant 20$, which implies $a_3 \leqslant 5$, yet Θ also contains $a_3 \geqslant 10$, leaving no possible value for a_3. Accordingly, any `Deposit` event in η for Alice at time 3 will create a violation at time 3, and this is known at time 2. Then, Alice's events in η is reported as a violation at time 2.

Applying Build to the assignment database for Bob at time 3, a constraint Θ is initialized to *true*. From α_b and its only match β_b, we have $true \rightarrow (a'_b = 3+5+b_3) \wedge (a'_b \leqslant 20)$. From γ_b and its only match δ_b, we have $true \rightarrow ((b'_b \geqslant 3) \wedge (b'_b \geqslant 5) \wedge (b'_b \geqslant b_3)) \wedge ((b'_b = 3) \vee (b'_b = 5) \vee (b'_b = b_3)) \wedge (b'_b \geqslant 10)$. From ω_b, there are no complete, ground, matching head assignments, so we add $\neg((a'_b = 3 + 5 + b_3) \wedge (a'_b \leqslant 20) \wedge (b_3 = 12))$. Now Θ is satisfiable, with the assignment $b_3 = 11$, $a'_b = 20$, $b'_b = 11$, and $c_b = 11$, so there is no violation of R at time 2 for Bob's events. ∎

The Build algorithm (Algorithm 7) starts with the assignment database $D_R(\eta)$ and a time instant t_0 (the current time), Line 1 initializes Θ as $true$, as the enactment η is not violating by default. Then, for each complete body assignment μ in $D_R(\eta)$ with constraints c_μ, there are two cases of the extension table for μ. In the first case (Line 3), μ has no complete, matching head assignment. Recall that c_μ are assumptions made about marked nulls that create μ if true. Accordingly, μ will not be a violation only if they are not necessary, i.e., their negation is satisfiable. We test for this case by adding $\neg c_\mu$ to Θ in Line 4. In the second case of the extension table for μ (Line 5), μ has one or more complete matching head assignments. Let (μ, μ_i, c_i), ..., (μ, μ_n, c_n) be the rows matching μ in the relevant extension table. Note that μ is a violation every c_i is unsatisfiable when c_μ is true. Thus, $c_\mu \to (c_1 \vee \cdots \vee c_n)$ is added to Θ on Line 6. Finally, Lines 7 and 8 add the requirement that unresolved timestamp variables and marked nulls time instants are greater than the current time. Generalizing this example, we state a theorem that indicates how the assignment database detects violations.

Theorem 5.3 : Let R be an acyclic set of rules with aggregation and η an enactment. Then, Build($D_R(\eta)$) is unsatisfiable if and only if η violates R.

Theorem 5.3 follows from Theorem 4.6 in the previous chapter and Lemma 5.2, which indicates that the rewriting of aggregation atoms by Rewrite preserves preserves constraints on the atoms' free variables imposed by the aggregation functions and time window. Then, the satisfiability of the output of Build faithfully indicates the presence of a violation. This satisfiability test remains decidable with PA formulas [71]. Accordingly, from Theorem 5.3, the output of Build is unsatisfiable exactly when the enactment violates the set of rules.

We have shown that for rules with aggregation, an assignment database is maintained with the Update, Update-E, and Chase-Agg algorithms and violations can be detected by applying Build to the assignment database, and reporting the satisfiability of Build's output. Thus, violations of an acyclic set of rules with aggreagtion are detected at the earliest possible time.

5.4 Related Work

Prior research on reasoning about aggregation in stream processing includes Datalog languages with aggregation, SQL-like stream queries, as well as temporal logics with aggregation, especially combined with first-order logic.

Some Datalog-like languages includes aggregation, often with restrictions on the monotonicity of event creation. whereas we do not require monotonicity of aggregation functions, because the generation of aggregation events is fixed by each window and the workflow assumption. Reference [72] uses Datalog with monotonic aggregation functions. Streamlog [30] uses Datalog with negation and arithmetic, but only for "strictly sequential" rules, i.e., the timestamp of head is greater every timestamp in the body. Reference [73] uses Datalog with negation and sets allowed as terms, but it is unclear how the *max* and *min* functions are computed; in Section 5.2, we show how these aggregation functions can be computed within Datalog. Dedalus [74] uses Datalog with negation, aggregation, and construct for disjunction called choice, but does not allow different time variables in the body of a rule, limiting the expressiveness of the language, and does not provide algorithms for rule evaluation. Finally, [75] uses Datalog with LTL operators and aggregation functions, with algorithms for rule evaluation. The main difference with our work is that we compare explicit time variables, with each other through inequalities, while the LTL operators in [75] make time implicit and difficult to compare due to scoping issues.

Also relevant are SQL-like stream queries, which are continually reevaluated with incoming data [76,77]. These queries use SQL-like syntax, with special syntax for windows, though their focus is query evaluation, i.e., the efficient computation of query results over the existing enactment, rather than early violation detection, which requires reasoning about the future of the enactment.

5.5 Chapter Summary

This chapter presents a syntax and semantics for time windows and aggregation functions in rules; this increases the expressiveness of rules, as it allows constraints on aggregate, quantitative values over time and multiple events. Then, we enable early violation detection for rules with aggregation in two ways: by rewriting them into Datalog$_{\mathbb{Z}}$ programs, which allows for generating aggregation events within the Datalog framework, and by extending the Build algorithm to handle aggregation functions by rewriting them in Presburger arithmetic. This shows that early violation detection remains possible for rules with aggregation.

Chapter 6

The Impossibility of Early Violation Detection

In this chapter, we show that it is undecidable whether or not a given set of rules is finitely satisfiable, i.e., whether or not there is a finite instance that satisfies it. This result indicates that early violation detection is impossible in general. Combined with Chapters 4 and 5, this indicates that acyclicity for a set of rules forms a tight boundary for solving early violation detection.

The chapter is organized as follows: Section 6.1 defines finite satisfiability and states the main result. Section 6.2 introduces a rule language Datalog$^+$ for technical development and reduces the empty-tape halting problem for Turing machines to the finite satisfiability of Datalog$^+$. In Sections 6.4, and 6.2, we show that finite satisfiability for a set of Datalog$^+$ rules reduces to finite satisfiability for a set of rules, completing the proof of the main result. Finally, in Sections 6.5 and 6.6 discuss related work and conclude the chapter.

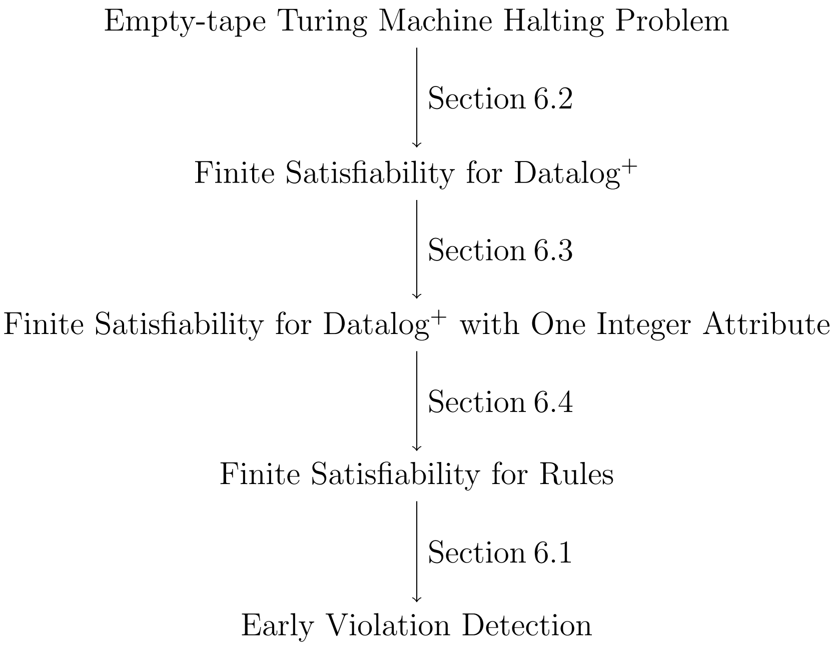
Empty-tape Turing Machine Halting Problem
Section 6.2
Finite Satisfiability for Datalog$^+$
Section 6.3
Finite Satisfiability for Datalog$^+$ with One Integer Attribute
Section 6.4
Finite Satisfiability for Rules
Section 6.1
Early Violation Detection

Figure 6.1: Structure of reductions

6.1 Early Violation Detection Solves Finite Satisfiability

In this section, we present the chapter's main result. We first give the necessary definitions for instances and finite satisfiability. We then state the main theorem and outline its proof. Finally, we present a corollary related to early violation detection.

Because we are interested in finite satisfiability of a set of rules, rather than the online early violation detection problem, we do not use enactments as the objects of rule satisfaction; instead, we say a set of rules is satisfied by an "instance". For a set S of event types, an *instance* of S is a mapping from each event type T in S to a possibly-infinite set of events of type T. For an instance I and an event type E, the *table* $I.E$ is the set of events, also called *tuples*, of type E in I. An instance *satisfies* a rule or set of rules with the same semantics as satisfaction for enactments in Chapter 2. A set of rules is *finitely satisfiable* if there is a finite instance that satisfies each rule in the set. The *finite satisfiability problem* is to decide whether a given set of rules is finitely satisfiable.

$$\boxed{\begin{aligned} & r_1 \ : \ \mathit{true} \rightarrow \texttt{Request}(\mathit{Alice})@x \\ & r_2 \ : \ \texttt{Request}(u)@y \rightarrow \texttt{Approve}(u)@y{+}1 \\ & r_3 \ : \ \texttt{Approve}(v)@y \rightarrow \texttt{Request}(v)@y{+}1 \end{aligned}}$$

Figure 6.2: Set of rules that is not finitely satisfiable

Example 6.1 : Not every set of rules is finitely satisfiable; consider the set of rules in Fig 6.2. Let I be an arbitrary instance satisfying $\{r_1, r_2, r_3\}$. Because r_1 has no atoms in the body, I must contain the event $\texttt{Request}(\mathit{Alice}, x)$ for some $x \in \mathbb{N}$; without loss of generality, we assume $x = 1$. Then, satisfying r_1 requires I contain $\texttt{Request}(\mathit{Alice}, 1)$. Applying r_2 to $\texttt{Request}(\mathit{Alice}, 1)$, I contains $\texttt{Approve}(\mathit{Alice}, 2)$. Applying r_3 to $\texttt{Approve}(\mathit{Alice}, 2)$, I contains $\texttt{Request}(\mathit{Alice}, 3)$. Then, applying r_2 to $\texttt{Request}(\mathit{Alice}, 3)$ indicates I contains $\texttt{Approve}(\mathit{Alice}, 4)$. Continuing in this manner, I contains $\texttt{Request}(\mathit{Alice}, 2n + 1)$ and $\texttt{Approve}(\mathit{Alice}, 2n + 2)$ for all $n \in \mathbb{N}$. In summary, the rule r_1 has no atoms in the body. This forces the instance to contain the event in the head of r_1. that comes from applying r_2 and r_3. That event initiates the infinite chain of events. Thus, I is infinite; because I is arbitrary, this shows no finite instance can satisfy $\{r_1, r_2, r_3\}$. ▮

Given these definitions, we can now state the main result of this chapter in Theorem 6.1.

Theorem 6.1 : Finite satisfiability for a set of rules is undecidable.

The remainder of this chapter is devoted to proving this theorem. We place this result in the context of early violation detection. Finite satisfiability for a set of rules is a special case of early violation detection in which the enactment is empty, as the empty enactment should be reported as a violation if and only if the set of rules is not finitely satisfiable. Then, an algorithm that decides if a violation exists also decides finite satisfiability. This leads to the following corollary of Theorem 6.1.

Corollary 6.2 : Early violation detection for a set of rules is impossible.

Our proof of Theorem 6.1 is structured as follows: in Section 6.2, we introduce a language called Datalog^{+} and show that the empty-tape halting problem reduces to finite satisfiability for a set of Datalog^{+} rules. Then, in Section 6.4, we show that there is a mapping between Datalog^{+} and rules that extends the undecidability result for Datalog^{+} to finite satisfiability for rules.

6.2 Datalog^{+}

In this section, we give definitions for the central notions of the reduction, including Datalog^{+}, the model of Turing machines and the empty-tape halting problem. Then, we construct a reduction from the empty-tape halting problem to finite satisfiability for Datalog^{+} rules.

For the technical development in the proof of Theorem 6.1, we define a new rule language called Datalog^{+}. Datalog^{+} has the same syntax and semantics as our rule language, except that it does not use the timestamp "@" syntax and each event type may have an arbitrary number of integer attributes, instead of only data attributes. Timestamp attributes take values from $\mathbb{N}$, with the standard ordering $<$, addition $+$, equality $=$, and non-equality $\neq$ predicates. An example of the syntax of Datalog^{+} is shown in Fig 6.3.

$$\boxed{\begin{array}{l} r_1 \,:\, \mathit{true} \rightarrow \texttt{Request}(\mathit{Alice}, x) \\ r_2 \,:\, \texttt{Request}(u, y) \rightarrow \texttt{Approve}(u, y{+}1) \\ r_3 \,:\, \texttt{Approve}(v, y) \rightarrow \texttt{Request}(v, y{+}1) \end{array}}$$

Figure 6.3: Set of Datalog^{+} rules that is not finitely satisfiable

A set of Datalog^{+} rules is satisfied by an instance, with the same notion of finite satisfiability, as defined for a set of rules. Given this definition, we state the following theorem, proven in the remainder of this section.

Theorem 6.3 : Finite satisfiability for a set of Datalog$^+$ rules is undecidable.

In Example 6.1, the event in the head of r_1 can be thought of as the input to a computation and chasing the rules r_2 and r_3 as an algorithm for the computation. In this view, whether or not the computation halts corresponds to whether or not the set of events required to satisfy the set of rules is finite or infinite. This suggests that the halting problem for Turing machine is reducible to the finite satisfiability problem for Datalog$^+$. To show this, we present the Turing machine as a formal model of computation, then show how to encode a Turing machine as a set of Datalog$^+$ rules in a way that matches its halting behavior to the finite satisfiability problem.

6.2.1 Turing Machines and the Empty-Tape Halting Problem

We use a standard definition of Turing machines and their computations [78]. A *deterministic Turing machine* M is a 5-tuple (Σ, Q, q_0, F, δ,) where

1. Σ is a non-empty set of *tape symbols*, one of which is the *blank symbol* $\textvisiblespace$,
2. Q is a non-empty set of *states*, with $q_0 \in Q$ as the *starting state* and $F \subseteq Q$ as a set of *halting states*, and
3. δ is a partial *transition function* from $Q \times \Sigma$ to $Q \times \Sigma \times \{L, R, \mathit{stay}\}$.

Note that no input alphabet is defined because we consider only computations where the input is empty. The behavior of a Turing machine is characterized as a sequence of configurations of the machine, snapshots of the machine's state and tape contents, recorded between each transition. For the Turing machine M, a *configuration* of M is a triple (w, j, s) where

- w is the *tape contents*, a finite word in the language $(\Sigma - \{\textvisiblespace\})^*$,
- j is the position of the read-head, such that $1 \leqslant j \leqslant |w| + 1$, indicating that the machine's read-head is reading the j-th symbol from the left in the non-blank portion

of the tape

- s is the state of the machine, i.e., a state in Q

Given any configuration C, the subsequent configuration is determined only by C and the machine's transition function, which maps each pair of a state and a tape symbol to a unique triple of a state, a tape symbol, and a direction. This makes our class of Turing machines *deterministic*. This definition only allows the machine to replace blank symbols with symbols from the tape alphabet to the right of the read-head's starting position, i.e., in one direction. This makes it a *one-way* Turing machine.

We consider only computations that start with the *empty-tape configuration*, i.e., with one ␣ symbol on the tape, the read-head at position 1 on the tape reading the ␣ symbol, and the machine in the start state q_0. The *empty-tape halting problem* asks: does a given deterministic one-way Turing machine halt after beginning with the empty-tape configuration? This problem is known to be undecidable [79].

6.2.2 Constructing a Set of Datalog$^+$ Rules for the Empty-Tape Halting Problem

We now encode the empty-tape halting problem for an arbitrary Turing machine as a set of Datalog$^+$ rules. The encoding is based on the idea that a Turing machine computation is a sequence of configurations. We represent each configuration as a block of rows in a table, one block per configuration. We use the following tables:

- `Config` encodes each of M's configurations, starting with the empty-tape configuration,

- `Next` helps link successive configuration blocks in `Config`, so that only successive configurations that are valid according to M's transition function are allowed,

- `Error` captures other conditions that `Config` and `Next` must satisfy to represent a valid computation of M, such as each configuration only having one machine state, with rules that trigger non-finite satisfiability if the computation is invalid.

Because the `Config` table stores each configuration in the empty-tape computation of M, it has an infinite number of rows if and only if the machine has an infinite computation on the empty tape. Additionally, M is deterministic, so the instance of `Config`, `Next`, and `Error` that satisfies $\textsc{Rules}(M)$ is unique given M. Then, $\textsc{Rules}(M)$ is finitely satisfiable if and only if M has a halting computation on the empty tape.

Transition Relation				
read	*state*	*write*	*nextState*	*dir*
␣	q_0	0	q_3	R
0	q_1	1	q_1	L
0	q_2	1	q_4	*stay*
␣	q_3	1	q_2	L
...				

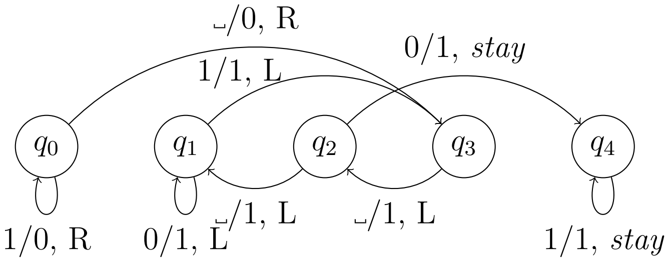

Table 6.1: Transitions for the Turing Machine M

Example 6.2 : Figure 6.1 shows the transition relation for a running example Turing machine M, along with a finite state machine diagram for M. In the transition relation for M, the first row indicates that when a ␣ symbol is read and the machine is in state q_0, the following configuration is achieved by writing a 0 symbol to the tape, changing the machine's state to q_1, and moving the read-head one tape cell to the right. The second row indicates that when a ␣ symbol is read and the machine is in state q_1, the machine writes a 1 symbol to the tape, maintains the state q_1, and moves the read-head one tape cell to the left. ▌

The `Config` event type holds the machine's tape contents and read-head position. It uses three attributes: *index*, *tape*, and *state*. The *index* is an integer attribute is unique for each row and organizes the configurations, with blocks of rows with consecutive indices encode an individual configurations. Finally, *tape* is a data attribute to hold tape symbols, and *state* is a data attribute to hold machine states.

`Next`	
index	*next*
0	2
1	3
2	5
3	6
4	7
5	8
6	9
7	10
8	11
. . .	

`Config`		
index	*tape*	*state*
0	#	␣
1	␣	q_0
2	#	␣
3	0	␣
4	␣	q_3
5	#	␣
6	0	q_2
7	1	␣
8	#	␣
...		

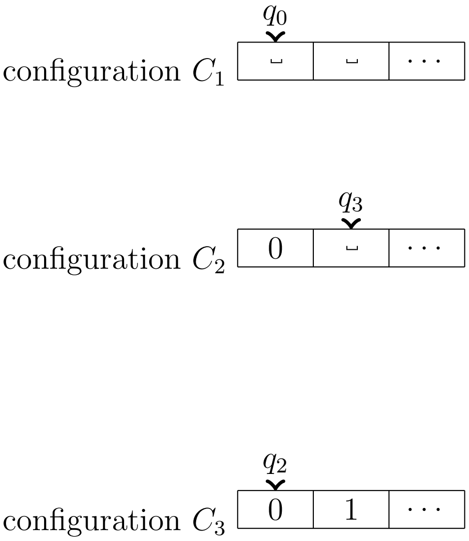

Figure 6.4: Four successive configurations of M in the `Config` and `Next` tables

Example 6.3 : Consider the Table 6.4, the row with index 1 represents the initial empty-tape configuration of a machine M, shown as configuration C_1. The *tape* column holds the tape's contents, initially the string ␣, and the read-head position, 1, and (starting) state q_0 is indicated by the row where the *state* column is not blank; as M is in state q_0 at the start of the computation. The following configuration, C_2, produced when M writes

Initialization Rules
true → Config(0,#,␣)
true → Config(1,␣,q_0)
true → Config(2,#,␣)
true → Next(0,1)
true → Next(1,3)

Figure 6.5: The Initialization Rules for M

0 to the tape, moves right, and changes from state q_0 to q_3, is shown in the rows of Config with indices 3 and 4. The rows with indices 6 and 7 represent the next configuration, C_3, when M writes 1 to the tape, moves left, and changes from state q_3 to q_2. ▮

The Next table indicates how consecutive configurations in a Turing machine computation are arranged in the Config table. An tuple (i, j) in the Next table means information at index i in Config is propagated to index j in Config for the subsequent configuration, assuming one exists. For the first three configurations shown in the Config table above, the Next table is shown in Table 6.4. Note that the arithmetic difference between the two columns can grow, because the length of the non-blank portion of the tape grows by one from the first configuration to the second.

The *initialization* rules (Fig. 6.5) require that the Config table has the initial empty-tape configuration, i.e., a configuration with an empty tape and a read-head index of 1, and that the Next table indicates the indices of the second configuration. Because the body of these rules is vacuously true, these tuples appear in all instances that satisfy Rules(M).

The following rules in Rules(M) encode the progression the Config and Next tables with respect to the machine's transition function. These include "normal" transitions to the right or left, where the machine's transition function is applied, and "extending" transitions to the right, when the machine advances into the blank portion of the tape and the tape's length is extended by one. Additionally, there is the "stay" rule, which encodes the machine's transition function when the machine does not move the read-head, and the "copy" rule shows how the Config and Next tables map one configuration to the next. Finally, they also include an "error" transition rule, which triggers an error if the machine transitions to the left when the read-head is at the leftmost tape cell.

The *normal left transition* rule encodes the leftward transition of the read-head. For each of the machine's transitions $\delta(a, s) = (b, s', L)$, the machine changes state from s to s' when reading tape symbol a, write a tape symbol b to the tape cell below the read head, and move the read head to the left. This rule applies when the Next table indicates that the row with index $x{-}1$ in Config corresponds to the row with index $y{-}1$ in Config in the machine's following configuration and the row with index x in Config has state s and is reading tape symbol a. The rule's head ensures that the row with index y in Config has state s' and writes tape symbol b to the tape cell below the read head, and the Next table is updated with rows $\texttt{Next}(x, y)$, $\texttt{Next}(x{+}1, y{+}1)$, and $\texttt{Next}(x{+}2, y{+}2)$. The rule is:

$$\begin{aligned} & \texttt{Config}(x{-}1, c, \llcorner\!\lrcorner), \texttt{Config}(x, a, s), \texttt{Config}(x+1, c', \llcorner\!\lrcorner), \\ \texttt{Next}(x{-}1, y{-}1) \rightarrow{} & \texttt{Config}(y{-}1, c, \llcorner\!\lrcorner), \texttt{Config}(y, b, \llcorner\!\lrcorner), \texttt{Config}(y+1, c', s'), \\ & \texttt{Next}(x, y), \texttt{Next}(x+1, y+1), \texttt{Next}(x+2, y+2) \end{aligned} \tag{6.1}$$

The *normal right transition* rule encodes the rightwards motion of the read head when the read head is not on the rightmost cell of the tape, using the same format as the left transition rule for each transition $\delta(a, s) = (b, s', R)$,

$$\begin{aligned}
&\texttt{Config}(x{-}1, c, \text{\textvisiblespace}), \texttt{Config}(x, a, s), \texttt{Config}(x+1, c', \text{\textvisiblespace}),\\
\texttt{Next}(x{-}1, y{-}1) \rightarrow\ &\texttt{Config}(y{-}1, c, \text{\textvisiblespace}), \texttt{Config}(y, b, \text{\textvisiblespace}), \texttt{Config}(y+1, c', s'),\\
&\texttt{Next}(x, y), \texttt{Next}(x+1, y+1), \texttt{Next}(x+2, y+2)
\end{aligned} \tag{6.2}$$

The *extending right transition rule* encodes the rightwards motion of the read head when the read head is on the rightmost cell of the tape. The machine performs a state change, writes a symbol to the tape cell below the read head, and moves the right, creating a new blank symbol in the read head's new position:

$$\begin{aligned}
&\texttt{Config}(x{-}1, c, \text{\textvisiblespace}), \texttt{Config}(x, a, s), \texttt{Config}(x+1, \#, \text{\textvisiblespace}),\\
\texttt{Next}(x{-}1, y{-}1) \rightarrow\ &\texttt{Config}(y{-}1, c, \text{\textvisiblespace}), \texttt{Config}(y, b, \text{\textvisiblespace}), \texttt{Config}(y+1, \text{\textvisiblespace}, s'),\\
&\texttt{Config}(y+2, \#, \text{\textvisiblespace}), \texttt{Next}(x, y), \texttt{Next}(x+1, y+1), \texttt{Next}(x+2, y+3)
\end{aligned} \tag{6.3}$$

The *stay transition* rule encodes the lack of movement of the read head for a stay transition, which initiates a state change and writes a symbol to the tape cell below the read head, leaving the position of the read head unchanged. For each transition rule $\delta(a, s) = (b, s', \mathit{stay})$, the rule is:

$$\begin{aligned}
&\texttt{Config}(x{-}1, c, \text{\textvisiblespace}), \texttt{Config}(x, a, s), \texttt{Config}(x+1, c', \text{\textvisiblespace}),\\
\texttt{Next}(x{-}1, y{-}1) \rightarrow\ &\texttt{Config}(y{-}1, c, \text{\textvisiblespace}), \texttt{Config}(y, b, s'), \texttt{Config}(y+1, c', \text{\textvisiblespace}),\\
&\texttt{Next}(x, y), \texttt{Next}(x+1, y+1), \texttt{Next}(x+2, y+2)
\end{aligned} \tag{6.4}$$

The *copy* rule prompts non-halting configurations in `Config` table to be copied into higher indices according to the `Next` table:

$$\begin{aligned}
&\texttt{Config}(x{-}1, c', \text{\textvisiblespace}), \texttt{Config}(x, c, \text{\textvisiblespace}), \texttt{Config}(x+1, c'', \text{\textvisiblespace}), \texttt{Next}(x, y)\\
&\qquad \rightarrow \texttt{Config}(y, c, \text{\textvisiblespace}), \texttt{Next}(x+1, y+1)
\end{aligned} \tag{6.5}$$

Finally, Error rules in $\textsc{Rules}(M)$ indicate when the tables fail to represent a halting computation of M. This may be because the instance does not obey the transition function or the initial configuration, or because the Config or Next tables have inconsistent data. The Error event type has a single attribute and is "thrown" when an inconsistency is detected by creating a tuple in the Error table, which is then propagated by the *propagate error* rule. to fill the Error table with an infinite number of tuples:

$$\texttt{Error}(x) \rightarrow \texttt{Error}(x+1) \tag{6.6}$$

The *erring left transition* rule encodes the impossibility of the read-head moving left when the read head is on the leftmost cell of the tape. Attempting to move to the left indicates the Turing machine is not performing a (valid) computation and inserts a row into the Error table, which is then propagated infinitely:

$$\texttt{Config}(x{-}1, \#, \textvisiblespace), \texttt{Config}(x, a, s), \texttt{Next}(x{-}1, y{-}1) \rightarrow \texttt{Error}(0) \tag{6.7}$$

The *same index, different symbols error* rule prevents one index in the Config table from being mapped to two distinct tape symbols. Each index corresponds to one tape cell in one configuration of a machine, thus the tape symbol must be unique. For each pair of unique tape symbols a and b in the machine's tape alphabet, we have the rule:

$$\texttt{Config}(x, a, s), \texttt{Config}(x, b, s') \rightarrow \texttt{Error}(0) \tag{6.8}$$

The *same index, different states* rule enforces that each index x in the Config table is mapped to at most one state. For each pair of unique states s and s', we have the rule:

$$\texttt{Config}(x, a, s), \texttt{Config}(x, b, s') \rightarrow \texttt{Error}(0) \tag{6.9}$$

In summary, given an arbitrary Turing machine M, the set $\textsc{Rules}(M)$ includes:

1. The *initialization* rules, and

2. Rules 6.1 through 6.4 for each of the machine's δ transitions,

3. Rules 6.5 through 6.9

6.2.3 The Construction is a Reduction

We now prove that $\textsc{Rules}(M)$ is finitely satisfiable if and only if M has a halting computation on the initial empty-tape. We first prove the forward direction, that if M has a halting computation on the initial empty-tape, then $\textsc{Rules}(M)$ is finitely satisfiable. Then, we prove the reverse direction.

Claim A: If M has a halting computation on the initial empty-tape, $\textsc{Rules}(M)$ is finitely satisfiable.

Proof: Assume M has a halting computation on the empty-tape. Then, there is a finite computation S of M such that $S(0)$ is the initial blank-tape configuration and the last configuration of S has a halting state. We construct an instance $db(S)$ with event types `Config`, `Next`, and `Error`.

For each configuration $S(i)$ in S, we include tuples in $db(S)$.`Config` and $db(S)$.`Next` using the following mapping from configurations to tuples:

Let $db(S)$.`Config` include the tuples in Table 6.2 to represent $S(0)$.

db(S).`Config`

index	tape	state
0	#	␣
1	␣	q_0
2	#	␣

Table 6.2: Tuples in $db(S)$.`Config` From Initial Configuration

Similarly, let $db(S)$.`Next` include the tuples in Table 6.3 to represent the transition from $S(0)$ to $S(1)$.

db(S).Next

index	next
0	2
1	3

Table 6.3: Tuples in $db(S)$.`Next` From Initial Configuration

Let $S(i).tape[j]$ be the symbol on the j^{th} position of the tape in $S(i)$. Let $|S(i)|$ be the length of the non-blank portion of the tape in the configuration $S(i)$. If $S(i)$ is mapped to the tuples in Tables 6.4 and 6.5 and these tuples are included in $db(S)$, then $S(i+1)$ (if it exists) is mapped to the tuples in Tables 6.6 and 6.7 and those tuples are included in $db(S)$.

To see the instance $db(S)$ is finite, consider the following: the computation S is halting, so the number of configurations in S is finite. Each configuration $S(i)$ is mapped to $|S(i)| + 2$ tuples, so the $db(S)$.`Config` and $db(S)$.`Next` tables each have a finite number of tuples. We do not add any tuples in $db(S)$.`Error`, so $db(S).Error$ is empty by construction. Each table in $db(S)$ is finite, so $db(S)$ is finite.

Now we show $db(S)$ satisfies $\textsc{Rules}(M)$. We do this by showing $db(S)$ satisfies each rule in $\textsc{Rules}(M)$. The construction of $db(S)$ directly includes the tuples required by the initial configuration rules as the tuples mapped from $S(0)$. For the normal transition left rule, assume $db(S)$ satisfies the body of the rule with tuples `Next`(x, y), `Config`$(x-1, c, \sqcup)$, `Config`(x, a, s), and `Config`$(x+1, c', \sqcup)$.

Because the indices in the `Config` events are consecutive, they are created by some configuration $S(i)$ in S. Because the transition rule requires that the state s is not

db(S).`Config`

index	tape	state
j	#	␣
$j+1$	$S(i).tape[1]$	q_0
$j+2$	$S(i).tape[2]$	q_0
...	...	...
$j+\lvert S(i)\rvert+\lvert S(i+1)\rvert+1$	#	␣

Table 6.4: Tuples in $db(S)$.`Config` from $S(i)$

db(S).`Next`

index	next
$j+\lvert S(i)\rvert+1$	$j+\lvert S(i)\rvert+\lvert S(i+1)\rvert+2$
$j+\lvert S(i)\rvert+2$	$j+\lvert S(i)\rvert+\lvert S(i+1)\rvert+3$
...	...

Table 6.5: Tuples in $db(S)$.`Next` from $S(i)$

halting, there is a configuration $S(i+1)$ following $S(i)$. Because $S(i)$ and $S(i+1)$ obey the Turing machine's transition function, $S(i+1)$ will have symbol c is at index $y{-}1$, symbol b at index y, and symbol c' at index $y+1$ on its tape, and the read-head of the machine will be at index $y{-}1$ with state s'. Additionally, $S(i)$ maps to a tuple in `Config` with index x so in $db(s)$, `Next` contains the tuple $(x{-}1, y{-}1)$. Then, $S(i+1)$ is mapped to the tuples `Config`$(y{-}1, c, s')$, `Config`$(y, b, \text{␣})$, and `Config`$(y+1, c', \text{␣})$, as well as `Next`$(x+1, y+1)$, `Next`(x, y), and `Next`$(x+2, y+2)$ in $db(S)$. Then, $db(S)$ contain the tuples expected by the head of the rule for the assumed tuples.

For the erring transition left rule: Because S is a valid computation in M, there are

db(S).Config

index	tape	state
$j + \lvert S(i)\rvert + 1$	$\#$	$\textvisiblespace$
$j + \lvert S(i)\rvert + 2$	$S(i+1).tape[1]$	$\textvisiblespace$
...	...	...
$j + \lvert S(i)\rvert + \lvert S(i+1)\rvert + 2$	$\#$	$\textvisiblespace$

Table 6.6: Tuples in $db(S)$.`Config` from $S(i+1)$

db(S).Next

index	next
$j + \lvert S(i)\rvert + 1$	$j + \lvert S(i)\rvert + \lvert S(i+1)\rvert + 2$
$j + \lvert S(i)\rvert + 2$	$j + \lvert S(i)\rvert + \lvert S(i+1)\rvert + 3$
...	...
$j + \lvert S(i)\rvert + \lvert S(i+1)\rvert + 2$	$j+\lvert S(i)\rvert+\lvert S(i+1)\rvert+\lvert S(i+2)\rvert+3$ (+4 if $S(i+2)$ extends the tape)

Table 6.7: Tuples in $db(S)$.`Next` from $S(i+1)$

no movements of the read-head to the left of the left end of the non-blank tape, so tuples reflecting such a transition will not be inserted into $db(S)$. Then, the body of this rule will never be triggered, so this rule is vacuously satisfied.

For the normal transition right right, the reasoning is identical to that of the normal transition left rule, but with `Config`$(y-1, c_1, \textvisiblespace)$ and `Config`$(y+1, c_2, s')$ instead of `Config`$(y-1, c_1, s_2)$ and `Config`$(y+1, c_2, \textvisiblespace)$. For the extending transition right rule, the reasoning is identical to that of the normal transition right rule, but with `Next`$(x+1, y+2)$ and `Next`$(x+2, y+3)$ instead of `Next`$(x+1, y+1)$ and `Next`$(x+2, y+2)$. For the stay rule, the reasoning is identical to that of the normal transition right rule, but with $\texttt{Config}(y{-}1, c_1, \llcorner\lrcorner)$ and $\texttt{Config}(y, b, s')$ instead of $\texttt{Config}(y{-}1, c_1, \llcorner\lrcorner)$ and $\texttt{Config}(y, b, \llcorner\lrcorner)$.

For the copy rule, assume $db(S)$ satisfies the body of the copy rule with tuples $\texttt{Next}(x, y)$, $\texttt{Config}(x{-}1, c'', \llcorner\lrcorner)$, $\texttt{Config}(x, c, \llcorner\lrcorner)$, and $\texttt{Config}(x + 1, c'', \llcorner\lrcorner)$. Let $S(i)$ be the configuration in S that maps to these tuples in $db(S)$. If $S(i)$ creates the tuple in $db(S)$.Config with x and $db(S)$.Next contains $(x{-}1, y{-}1)$, then (x, y) and $(x + 1, y + 1)$ are corresponding tape positions in $S(i)$ and $S(i + 1)$. Because $S(i)$ and $S(i + 1)$ are consecutive configurations in S, c will appear at tape position y in $S(i + 1)$. By construction, $db(S)$ will contain the tuples $\texttt{Config}(y, c)$ and $\texttt{Next}(x + 1, y + 1)$. Then the tables $db(S)$.Config and $db(S)$.Next will contain the tuples expected by the rule for these tuples.

Now we show the instance $db(S)$ satisfies the rules regarding the Error table. Because no tuple is added to the Error table in the construction of $db(S)$, the propagate error rule is vacuously satisfied.

For the erring transition left rule, S is a valid computation in M, so there are no transitions to the left of the left end of the non-blank tape, so the body of this rule is never triggered, so this rule is vacuously satisfied.

For the same index, different symbols error rule, In the construction of $db(S)$, each index used in $db(S)$.Config corresponds to one non-blank tape cell in exactly one configuration in S or to the left-end and respectively right-end blank tape cell in consecutive configurations in S. Then, for each index in $db(S)$.Config associated with a non-blank symbol, there is exactly one tape cell that generates a tuple with that index, so the tape symbol will be unique. In the case that the index is associated with a blank symbol from the left-end and right-end of the tape in consecutive configurations, the tape symbol is the blank symbol, so the tape symbol associated with the index is unique.

For the same index, different states error rule, in the construction of $db(S)$, each index used in $db(S)$.`Config` corresponds to one non-blank tape cell in exactly one configuration in S or to the left-end and respectively right-end blank tape cell in consecutive configurations in S. Then, for each index in $db(S)$.`Config` associated with a state, there is exactly one tape cell that generates a tuple with that index, meaning the tape symbol will be unique. In the case that the index is associated with a blank symbol from the left-end and right-end of the tape in consecutive configurations, the state symbol is the blank symbol, so the state associated with the index is unique.

The above cases show that $db(S)$ satisfies every rule in RULES(M). This, along with the fact that $db(S)$ is finite, shows there is a finite instance satisfying RULES(M). This concludes the proof of the ⇐ direction. ▮

Now we show if RULES(M) is finitely satisfiable, then M has a halting computation on the empty-tape. We do this by constructing a computation S in M from a finite instance I satisfying RULES(M).

Claim B: If RULES(M) is finitely satisfiable, M has a halting computation on the initial empty-tape.

Proof: Assume RULES(M) is finitely satisfiable. Then, there is a finite instance I that satisfies RULES(M). Because I satisfies `Error`(x) → `Error`($x+1$) and I is finite, no *error* rules are not triggered by I. Then, the table I.`Error` is empty. Then, the *same index, different symbols* and *same index, different states* rules are not triggered by I. Then, every index in I.`Config` is paired with at most one state and at most one tape symbol.

In the following, we show I encodes a halting computation of M by showing consecutive sets of tuples in I correspond to consecutive configurations in a computation in M.

Let `tm` be a mapping from sets of consecutive tuples in I to configurations of M. For each set of consecutive tuples in I.`Config` with indices $[i, i+k]$ such that indices $i-1$ and $i+k+1$ have # in the *tape* column and no other indices in $[i, i+k]$ have # in the *tape* column, let $tm(i, i+k)$ be the configuration of M such that

- the tape contents are the sequence of symbols in the *tape* column, ordered by the indices
- the read-head position is the position on the tape (relative to i) of the state symbol that is not "-" in the state column
- the machine state is the state symbol that is not "-" in the state column

Now we show that I is a valid computation in M. Let $tm(i, j)$ be a configuration of M that does not have a halting state and I contain $\texttt{Next}(i, j)$ and $\texttt{Next}(i+1, j+1)$. We show that for some $k \geqslant j$, $\texttt{tm}(i, j)$ and $\texttt{tm}(j, k)$ are consecutive configurations in a computation of M and I contains $\texttt{Next}(j, k)$ and $\texttt{Next}(j+1, k+1)$. Let $i+w$ be the position of the machine's state in tuples with indices in the range $[i, j]$. There are two cases of $i+w$.

Case 1: $i+w = i+1$, i.e., the tuple in $I.\texttt{Config}$ with index $i+1$ has a machine state. Because $\texttt{tm}(i, j)$ is not halting, there is a configuration of M following $\texttt{tm}(i, j)$. Then, one of the transition rules applies with $x = i+1$ and I contains $\texttt{Next}(i+1, j+1)$, $\texttt{Next}(i+2, j+2)$, and $\texttt{Next}(i+3, j+3)$ (or $\texttt{Next}(i+3, j+4)$ in the case that the extending right transition rule applies). Then, the copy rule can be applied with index pairs $(x = i+3, y = j+3)$. Then, the copy rule can be applied with index pairs $(x = i+4, y = j+4), \ldots, (x = j-1, y = k)$ for some k because if the copy rule applies with $i+2$, the copy rule requires that $\texttt{Next}(i+3, j+3)$ be present, which will trigger the copy rule with $i+4$ unless a non-"-" state appears. In this case, the machine's state for $\texttt{tm}(i, j)$ is at index $i+1$, so the next non-"-" state for not until at least $j+1$. Then, the copy rule copes all tuples with indices in the range $[i+3, j]$ in $\texttt{Config}$, along with $\texttt{Next}(j, k)$ and $\texttt{Next}(j+1, k+1)$ for some k.

Case 2: $i+w > i+1$, i.e., the tuple in $I.\texttt{Config}$ with index $i+1$ does not have a machine state. Then, the copy rule applies to tuples with (x, y) values: $(x = i+1, y =$

$j+1), (x = i+2, y = j+2), ...(x = i+w-2, y = j+w-2)$, because each time the copy rule applies with some $(x = a, y = b)$, the copy rule requires the presence of $\texttt{Next}(a+1, b+1)$. This ensures the copy rule is triggered for the value $a + 1$ unless a non-“-” state appears in the tuple with index $a + 2$. In this case, the next non-“-’ ’state is in the tuple with index $i + w$. Then the Next table must have the row $\texttt{Next}(i + w-1, j + w-1)$. Because the tuple with index $i + w$ contains a state and I contains $\texttt{Next}(i + w-1, j + w-1)$, a transition rule can be applied.

There is a configuration of M following $tm(i, j)$. Then one of the transition rules is triggered with $x = i+w$, ensuring the tuples with indices $i+w-1$, $i+w$, and $i+w+1$, are mapped to the appropriate symbols and state in tuples with indices $j + w + 1$, $j + w + 2$, and $j + w + 3$ in Config. Finally, applying the transition rule with $x = i + w$ requires the instance to have $\texttt{Next}(i+w, j+w)$ and $\texttt{Next}(i+w+1, j+w+1)$, and $\texttt{Next}(i+w+2, j+w+2)$ (or $\texttt{Next}(i + w + 2, j + w + 3)$, if the extending right transition rule applies). Then, the copy rule applies with $x = i + w + 2$, $x = i + w + 3$, $\ldots, x = j + w-1$, because when the copy rule applies with x, the tuple $\texttt{Next}(x + 1, y + 1)$ must be present, ensuring that the copy rule is triggered with $x + 1$ unless a non-“␣” state appears. In this case, the first non-“-” state is no earlier than the tuple in Config with index $j + w$.

Given that I encodes a computation of M, we show that this computation is finite and halting. For sake of contradiction, assume that M does not have a halting computation on the empty-tape. We show that this implies I is infinite, which contradicts the fact that I is finite.

Because M does not have a halting computation on the empty-tape, there is an infinite monotonically increasing sequence K of indices in I.Config such that $\texttt{tm}(k(i), k(i + 1))$ is a configuration in a computation of M on the empty-tape and I contains $\texttt{Next}(k(i + 1), k(i + 2))$ and $\texttt{Next}(k(i + 1) + 1, k(i + 2) + 1)$. We prove this by strong induction on the index i.

Base Case: Let $k(1) = 0$, $k(2) = 2$. Because I satisfies $\textsc{Rules}(M)$, the `Config` table in I satisfies the Initialization rule. Then, $tm(0, 2)$ corresponds to the initial configuration to M on the empty-tape and I contains $\texttt{Next}(0, 2)$ and $\texttt{Next}(1, 3)$.

Inductive Hypothesis: Assume there are some indices $k(i)$, $k(i+1)$ such that $tm(k(i), k(i + 1))$ is a configuration in a computation of M on the empty-tape and I contains $\texttt{Next}(k(i + 1), k(i + 2))$ and $\texttt{Next}(k(i + 1) + 1, k(i + 2) + 1)$.

With the assumption that $\texttt{tm}(k(i), k(i + 1))$ does not have a halting state, we know there is some k' such that $tm(k(i + 1), k')$ follows $tm(k(i), k(i + 1))$ and I contains $\texttt{Next}(k(i + 1), k')$ and $\texttt{Next}(k(i + 1) + 1, k' + 1)$. Let $k(i + 2) = k'$. Then, I has a tuple for each element of the infinite monotonically increasing sequence K, so I is infinite. This contradicts the assumption that I is finite. Then, we conclude M has a halting computation on the empty-tape. This concludes the proof of the ⇒ direction. ▮

This concludes both directions of logical implication. Theorem 6.3 follows from the undecidability of the empty-tape halting problem [79] and this reduction from the empty-tape halting problem is reduced to the finite satisfiability problem for rules. In the next section, the encoding is shown to hold when Datalog$^+$ is limited to only one integer attribute.

6.3 Datalog$^+$ with One Integer Attribute

It is natural to ask whether the undecidability result holds for events with one integer attribute, as an increasing number of integer attributes suggests an increase in the scale or complexity of enactments and sets of rules. In this section, we show that finite satisfiability for Datalog$^+$ rules with exactly one integer attribute is undecidable, by showing all events in the encoding $\textsc{Rules}(M)$ can be simulated with events with one integer attribute. Then, the proof of Theorem 6.1 holds with this restriction.

Theorem 6.4 : Finite satisfiability for a set of Datalog$^+$ rules with one integer attribute is undecidable.

Proof: Given Theorem 6.3, it is sufficient to show Theorem 6.4 by showing that `Config`, `Next`, and `Error` can be simulated with event types with one integer attribute. The simulation works as follows: let $X(a_1, \ldots, a_n)$ be an event types with n attributes Then, we can create n new event types X_1, X_2, $\ldots$, X_n, each with one integer attribute. The integer attribute of X_i is a_i; note that a data attribute a_i in X can be simulated by an integer attribute in X_i, as the integers have the same cardinality as the data domain, as well as the equality and non-equality predicates. The second attribute of each X_i is a new data attribute *join*, which is used to join the X_i's to simulate the X event type. We show how each of the four event types in $\textsc{Rules}(M)$ is simulated by this encoding.

Recall that the `Config` event type has three attributes: *index*, *tape*, and *state*. Then, we can simulate `Config` events using the following three event types with the new attribute *join*: C_1(*index*, *join*), C_2(*tape*, *join*), and C_3(*state*, *join*). Then, each event atom $\texttt{Config}(x, t, s)$ in a rule in $\textsc{Rules}(M)$ is replaced with $C_1(x, i)$, $C_2(t, i)$, $C_3(s, i)$, where i is a fresh variable in each replacement of a `Config` atom. Similarly, `Next`(*index*, *next*) can be simulated with event types $\texttt{Next}_1$(*index*, *join*) and $\texttt{Next}_2$(*next*, *join*). For rules with more than one `Config` or `Next` atom, we use different variables $i_1, i_2, \ldots$ for the *join* attribute for each replacement and include non-equality atoms between these join variables. For example, the copy rule:

$$\begin{aligned} &\texttt{Config}(x{-}1, c', \textvisiblespace), \texttt{Config}(x, c, \textvisiblespace), \texttt{Config}(x+1, c'', \textvisiblespace), \texttt{Next}(x, y) \\ &\qquad \rightarrow \texttt{Config}(y, c, \textvisiblespace), \texttt{Next}(x+1, y+1) \end{aligned} \tag{6.10}$$

becomes

$$\begin{aligned}\texttt{Config}_1(x{-}1, i_1), \texttt{Config}_2(c', i_1), \texttt{Config}_3(\text{␣}, i_1),\\ \texttt{Config}_1(x, i_2), \texttt{Config}_2(c, i_2), \texttt{Config}_3(\text{␣}, i_2),\\ \texttt{Config}_1(x{+}1, i_3), \texttt{Config}_2(c'', i_3), \texttt{Config}_3(\text{␣}, i_3),\\ \texttt{Next}_1(x, i_4), \texttt{Next}_2(y, i_4)\\ \rightarrow \texttt{Config}_1(y, i_5), \texttt{Config}_2(c, i_5), \texttt{Config}_3(\text{␣}, i_5),\\ \texttt{Next}_1(x{+}1, i_6), \texttt{Next}_2(y{+}1, i_6)\end{aligned} \tag{6.11}$$

Finally, note that the `Error` event type already has one integer attribute and no data attributes, so it is trivial to simulate.

Given a Turing machine M, we can construct a set of rules using the new event types. Because the new event types simulate the original tuples in Rules(M), the proof of Theorem 6.3 applies. ▮

6.4 Proof of Theorem 6.1

It remains to be shown that the undecidability of finite satisfiability for Datalog$^+$ indicates the same result for the original rule language, completing the proof of Theorem 6.1. The previous section proves undecidability for Datalog$^+$ with one integer attribute, so it is sufficient to show that there is a mapping from this language to the original rule language that preserves finite satisfiability. We show this mapping exists in Lemma 6.5.

Lemma 6.5 : Let R be a set of rules in Datalog$^+$ with one integer attribute. There is a set R' of constraints rule that is finitely satisfiable if and only if R is finitely satisfiable.

Proof: Without loss of generality, let $E(d_1, \ldots, d_n, t)$ be an event type in R, where t is the integer attribute. Note that this event type generates events with exactly one timestamp. Then, there is an equivalent event type $E(d_1, \ldots, d_n)$ for rules as events for E

have a timestamp, e.g., $E(Alice)@10$, and event atoms for E have a timestamp variable, e.g., $E(x, y, z)@t$. Thus, $E(d_1, \ldots, d_n, t)$ in Datalog$^+$ and $E(d_1, \ldots, d_n)$ in rules generate identical events and have identical event atoms, modulo timestamp syntax. Then, to build R', replace each event atom $E(a_1, \ldots, a_n, t)$ in R with $E(a_1, \ldots, a_n)@t$, producing a rule that is satisfied by the same instances as R. Thus, R is finitely satisfiable if and only if R' is finitely satisfiable. ▮

Then, the undecidability of finite satisfiability for a set of Datalog$^+$ rules with one integer attribute implies the same result for a set of rules.

6.5 Related Work

Finite satisfiability for first-order formulas, a superclass of rules, is undecidable, which can also be proven by reduction from the halting problem for empty-tape Turing machines [80]. A problem on bounding the depth of recursion for Datalog programs is also shown to be undecidable by reduction from Turing machines [81]. Reference [82] indicates undecidability of finite satisfiability for source-to-target tuple-generating dependencies, a class of formulas similar to rules where a partition exists between event types that appear in the body and event types that appear in the head. Deciding finite satisfiability for rules is similar to deciding the implication problem for constrained tuple-generating dependencies, for which chase techniques have been applied [83, 84].

6.6 Chapter Summary

In this chapter, we showed that the general problem of early violation detection for a set of rules is impossible, given the undecidability of finite satisfiability. It remains to be seen if finite satisfiability for rules with additional restrictions, such as using fewer data attributes, is still unsolvable. The chained encoding of the Turing machine tape and successive potentially-infinite configurations suggests a similar reduction with only unary event types is not possible.

Chapter 7

Conclusion

Driven by the demand for efficient and effective monitoring of event streams, we investigate early violation detection for business rules. Enforcing these constraints at runtime, rather than at design-time, allows enables the enforcement of business rules in event-based systems, while avoiding intractable design-time analysis. In this dissertation, we show that early violation detection for sets of rules is unsolvable in general, indicating that more research is needed to identify tractable cases. We also contribute algorithms to solve restricted cases of the early violation detection problem, relying on techniques from business process management, automated reasoning, and database systems. We study the translation of dataless rules to LTL formulas, improving the output size of the best known translation for two subclasses of rules. We consider the problem for individual rules and acyclic sets of rules with data, applying a chase process and satisfiability testing, then add aggregation functions on time windows to the rule language. These techniques are novel in their application to early violation detection, especially in the context of quantitative time constraints and incremental monitoring of event streams.

More work is needed to understand violation detection problems. This includes the study of richer classes of temporal constraints, beyond the gap constraints considered in this dissertation. Additionally, rules lack some features common in natural language and compliance regulations that may be useful in modeling real-world constraints, e.g., negation for modeling the absence of events and disjunction for modeling choice or multiple possibilities. More work is needed to determine how these features affect the complexity of the early violation detection problem.

Also, our techniques consider only perfect data and whether or not a violation is certain. Realistic event streams may also contain some noise and uncertainty, so reporting violations probabilistically and relaxing assumptions about the event stream's quality, e.g., allowing out-of-order events, deserve further study. Finally, the undecidability proof suggests that the border between solvable and unsolvable problems can be made sharper with respect to the number and type of event attributes. In summary, our insights improve the understanding of the early violation detection problem and may guide the design of rule monitors for event-based systems, but more work is needed to understand the full range of effective early violation detection.

Bibliography

[1] Y. Zhou, P. K. Chrysanthis, V. Gulisano, and E. T. Zacharatou, *Proceedings of the 16th ACM International Conference on Distributed and Event-based Systems, DEBS 2022, Copenhagen, Denmark, June 27-30, 2022.* Association for Computing Machinery, 2022.

[2] E. Damaggio, A. Deutsch, R. Hull, and V. Vianu, *Automatic verification of data-centric business processes*, in *BPM 2011*, pp. 3–16, 2011.

[3] D. Calvanese, G. D. Giacomo, and M. Montali, *Foundations of data-aware process analysis: A database theory perspective*, in *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pp. 1–12, 2013.

[4] Y. Du, W. Tan, and M. Zhou, *Timed compatibility analysis of web service composition: A modular approach based on petri nets*, *IEEE Transactions on Automation Science and Engineering* **11** (2014), no. 2 594–606.

[5] T. Bultan, J. Su, and X. Fu, *Analyzing conversations of web services*, *IEEE Internet Computing* **18** (2006), no. 10.

[6] M. Krotsiani, C. Kloukinas, and G. Spanoudakis, *Cloud certification process validation using formal methods*, in *ICSOC 2017*, pp. 65–79, Springer, 2017.

[7] A. Pnueli, *The temporal logic of programs*, in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pp. 46–57, IEEE, 1977.

[8] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, *Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, vol. 6418. Springer, 2010.

[9] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, *Introduction to runtime verification*, in *Lectures on Runtime Verification*, pp. 1–33. Springer, 2018.

[10] E. Bartocci, Y. Falcone, and G. Reger, *International competition on runtime verification (crv)*, in *Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25*, pp. 41–49, Springer, 2019.

[11] L. Baresi and S. Guinea, *Towards dynamic monitoring of ws-bpel processes*, in *International Conference on Service-Oriented Computing*, pp. 269–282, Springer, 2005.

[12] R. Fernando, R. Ranchal, B. Bhargava, and P. Angin, *A monitoring approach for policy enforcement in cloud services*, in *CLOUD 2017*, pp. 600–607, IEEE, 2017.

[13] O. Kupferman and M. Y. Vardi, *Model checking of safety properties*, *Formal methods in system design* **19** (2001) 291–314.

[14] D. Giannakopoulou and K. Havelund, *Automata-based verification of temporal properties on running programs*, in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pp. 412–416, IEEE, 2001.

[15] A. Bauer, M. Leucker, and C. Schallhart, *Runtime verification for ltl and tltl*, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **20** (2011), no. 4 1–64.

[16] R. De Masellis, F. M. Maggi, and M. Montali, *Monitoring data-aware business constraints with finite state automata*, in *Proceedings of the 2014 International Conference on Software and System Process*, pp. 134–143, 2014.

[17] A. Bauer, J.-C. Küster, and G. Vegliach, *From propositional to first-order monitoring*, in *International Conference on Runtime Verification*, pp. 59–75, Springer, 2013.

[18] L. T. Ly, F. M. Maggi, M. Montali, S. Rinderle-Ma, and W. M. Van Der Aalst, *Compliance monitoring in business processes: Functionalities, application, and tool-support*, *Information systems* **54** (2015) 209–234.

[19] S. Ceri, G. Gottlob, L. Tanca, *et. al.*, *What you always wanted to know about datalog(and never dared to ask)*, *IEEE transactions on knowledge and data engineering* **1** (1989), no. 1 146–166.

[20] I. Mackey and J. Su, *A rule-based constraint language for event streams*, in *4th International Workshop on the Resurgence of Datalog in Academia and Industry*, pp. 145–150, 2022.

[21] P. Z. Revesz, *A closed form for datalog queries with integer (gap)-order constraints*, *Theoretical Computer Science* **116** (1993), no. 1 117–149.

[22] M. Y. Vardi and P. Wolper, *An automata-theoretic approach to automatic program verification*, in *Proceedings of the First Symposium on Logic in Computer Science*, pp. 322–331, IEEE Computer Society, 1986.

[23] F. Laroussinie, N. Markey, and P. Schnoebelen, *Temporal logic with forgettable past*, in *Logic in Computer Science*, pp. 383–392, IEEE, 2002.

[24] P. Gastin and D. Oddoux, *Ltl with past and two-way very-weak alternating automata*, in *International Symposium on Mathematical Foundations of Computer Science*, pp. 439–448, Springer, 2003.

[25] J. A. W. Kamp, *Tense logic and the theory of linear order.* University of California, Los Angeles, 1968.

[26] A. Rabinovich, *A proof of kamp's theorem*, *Logical Methods in Computer Science* **10** (2014).

[27] A. Cimatti, M. Roveri, and D. Sheridan, *Bounded verification of past ltl*, in *International Conference on Formal Methods in Computer-Aided Design*, pp. 245–259, Springer, 2004.

[28] O. Lichtenstein, A. Pnueli, and L. Zuck, *The glory of the past*, in *Workshop on Logic of Programs*, pp. 196–218, Springer, 1985.

[29] G. B. Dantzig, *Fourier-motzkin elimination and its dual*, tech. rep., Standford University, Department of Operations Research, 1972.

[30] C. Zaniolo, *Logical foundations of continuous query languages for data streams*, in *International Datalog 2.0 Workshop*, pp. 177–189, Springer, 2012.

[31] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, *On the temporal analysis of fairness*, in *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 163–173, 1980.

[32] D. M. Gabbay, *Expressive functional completeness in tense logic (preliminary report)*, in *Aspects of philosophical logic*, pp. 91–117. Springer, 1981.

[33] I. Hodkinson, *Expressive completeness of until and since over dedekind complete linear time*, *Modal logic and process algebra* (1995) 171–185.

[34] M. Pesic, H. Schonenberg, and W. M. Van der Aalst, *Declare: Full support for loosely-structured processes*, in *EDOC 2007*, pp. 287–287, IEEE, 2007.

[35] F. M. Maggi, M. Montali, M. Westergaard, and W. M. Van Der Aalst, *Monitoring business constraints with linear temporal logic: An approach based on colored automata*, in *International Conference on Business Process Management*, pp. 132–147, Springer, 2011.

[36] K. Havelund and D. Peled, *Efficient runtime verification of first-order temporal properties*, in *Int. Symp. on Model Checking Software*, pp. 26–47, Springer, 2018.

[37] M. Montali, M. Pesic, W. M. v. d. Aalst, F. Chesani, P. Mello, and S. Storari, *Declarative specification and verification of service choreographies*, *ACM Transactions on the Web (TWEB)* **4** (2010), no. 1 1–62.

[38] M. Westergaard and F. M. Maggi, *Looking into the future*, in *OTM Confederated International Conferences: On the Move to Meaningful Internet Systems*, pp. 250–267, Springer, 2012.

[39] M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. van der Aalst, *Monitoring business constraints with the event calculus*, *TIST 2013* **5** (2013), no. 1 17.

[40] J. Li and K. Y. Rozier, *Mltl benchmark generation via formula progression*, in *International Conference on Runtime Verification*, pp. 426–433, Springer, 2018.

[41] D. Basin, B. N. Bhatt, and D. Traytel, *Almost event-rate independent monitoring of metric temporal logic*, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 94–112, Springer, 2017.

[42] O. Maler, D. Nickovic, and A. Pnueli, *Real time temporal logic: Past, present, future*, in *International Conference on Formal Modeling and Analysis of Timed Systems*, pp. 2–16, Springer, 2005.

[43] R. Dechter, I. Meiri, and J. Pearl, *Temporal constraint networks*, *Artificial intelligence* **49** (1991), no. 1-3 61–95.

[44] L. Hunsberger and R. Posenato, *Sound-and-complete algorithms for checking the dynamic controllability of conditional simple temporal networks with uncertainty*, in *25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[45] C. Combi, R. Posenato, L. Viganò, and M. Zavatteri, *Conditional simple temporal networks with uncertainty and resources*, *Journal of Artificial Intelligence Research* **64** (2019) 931–985.

[46] M. Franceschetti, R. Posenato, C. Combi, and J. Eder, *Dynamic controllability of parameterized cstnus*, in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pp. 965–973, 2023.

[47] J. Köpke, J. Eder, and J. Su, *Gsm+ t: A timed artifact-centric process model*, in *25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*, pp. 0–0, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[48] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases.* Addison-Wesley, 1995.

[49] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, *Data exchange: semantics and query answering*, *Theoretical Computer Science* **336** (2005), no. 1 89–124.

[50] F. Chen and G. Roşu, *Parametric trace slicing and monitoring*, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 246–261, Springer, 2009.

[51] G. Siqueria, "Log generator." `https://github.com/GabrielSiq/LogGenerator`, 2020.

[52] I. Mackey, *multi-rule monitor*, 2023.

[53] L. De Moura and N. Bjorner, *Z3: An efficient smt solver*, in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[54] L. De Moura and N. Bjørner, *Z3Py: Python Interface to the Z3 Theorem Prover*. Microsoft Research, 2021.

[55] F. M. Maggi, M. Montali, M. Westergaard, and W. M. Van Der Aalst, *Monitoring business constraints with linear temporal logic: An approach based on colored automata*, in *International Conference on Business Process Management*, pp. 132–147, Springer, 2011.

[56] F. M. Maggi, M. Westergaard, M. Montali, and W. M. van der Aalst, *Runtime verification of ltl-based declarative process models*, in *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers 2*, pp. 131–146, Springer, 2012.

[57] F. M. Maggi, M. Westergaard, M. Montali, and W. M. van der Aalst, *Runtime verification of ltl-based declarative process models*, in *International Conference on Runtime Verification*, pp. 131–146, Springer, 2011.

[58] C. Dousson and P. Le Maigat, *Chronicle recognition improvement using temporal focusing and hierarchization.*, in *IJCAI*, vol. 7, pp. 324–329, 2007.

[59] F. M. Maggi, M. Montali, and U. Bhat, *Compliance monitoring of multi-perspective declarative process models*, in *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 151–160, IEEE, 2019.

[60] R. De Masellis and J. Su, *Runtime enforcement of first-order ltl properties on data-aware business processes*, in *International Conference on Service-Oriented Computing*, pp. 54–68, Springer, 2013.

[61] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard, *Quantified event automata: Towards expressive and efficient runtime monitors*, in *International Symposium on Formal Methods*, pp. 68–84, Springer, 2012.

[62] J. Chomicki, *Efficient checking of temporal integrity constraints using bounded history encoding*, *ACM Transactions on Database Systems* **20** (1995), no. 2 149–186.

[63] S. Hallé and R. Villemaire, *Runtime enforcement of web service message contracts with data*, *IEEE Transactions on Services Computing* **5** (2012), no. 2 192–206.

[64] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu, *Monitoring metric first-order temporal properties*, *Journal of the ACM (JACM)* **62** (2015), no. 2 1–45.

[65] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, *Maintaining views incrementally*, *ACM SIGMOD Record* **22** (1993), no. 2 157–166.

[66] F. Afrati, C. Li, and V. Pavlaki, *Data exchange in the presence of arithmetic comparisons*, in *Proceedings of the 11th international conference on extending database technology: Advances in database technology*, pp. 487–498, 2008.

[67] B. ten Cate, P. G. Kolaitis, and W. Othman, *Data exchange with arithmetic operations*, in *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 537–548, 2013.

[68] G. De Giacomo, R. De Masellis, and M. Montali, *Reasoning on ltl on finite traces: Insensitivity to infiniteness*, in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, 2014.

[69] F. Chiariello, F. M. Maggi, and F. Patrizi, *From ltl on process traces to finite-state automata*, .

[70] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, *Complexity and expressive power of logic programming*, *ACM Computing Surveys (CSUR)* **33** (2001), no. 3 374–425.

[71] R. Stansifer, *Presburger's article on integer arithmetic: Remarks and translation*, tech. rep., Cornell University, 1984.

[72] A. Shkapsky, M. Yang, and C. Zaniolo, *Optimizing recursive queries with monotonic aggregates in deals*, in *2015 IEEE 31st International Conference on Data Engineering*, pp. 867–878, IEEE, 2015.

[73] A. Mohapatra and M. Genesereth, *Incremental maintenance of aggregate views*, in *International Symposium on Foundations of Information and Knowledge Systems*, pp. 399–414, Springer, 2014.

[74] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears, *Dedalus: Datalog in time and space*, in *International Datalog 2.0 Workshop*, pp. 262–281, Springer, 2010.

[75] L. Bellomarini, M. Nissl, and E. Sallinger, *Monotonic aggregation for temporal datalog.*, in *RuleML+ RR (Supplement)*, 2021.

[76] A. Arasu, S. Babu, and J. Widom, *The cql continuous query language: semantic foundations and query execution*, *The VLDB Journal* **15** (2006) 121–142.

[77] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, *Stream: The stanford data stream management system*, in *Data Stream Management*, pp. 317–336. Springer, 2016.

[78] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the theory of NP-Completeness.* Freeman, 1979.

[79] T. D. Banitt, *The halting problem*, *Macalester Journal of Philosophy* **5** (2010), no. 1 4.

[80] L. Libkin, *Elements of finite model theory*, vol. 41. Springer, 2004.

[81] G. G. Hillebrand, P. C. Kanellakis, H. G. Mairson, and M. Y. Vardi, *Undecidable boundedness problems for datalog programs*, *The Journal of logic programming* **25** (1995), no. 2 163–190.

[82] P. G. Kolaitis, J. Panttaja, and W.-C. Tan, *The complexity of data exchange*, in *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 30–39, 2006.

[83] M. J. Maher and D. Srivastava, *Chasing constrained tuple-generating dependencies*, in *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 128–138, 1996.

[84] D. Dou and S. Coulondre, *A sound and complete chase procedure for constrained tuple-generating dependencies*, *Journal of Intelligent Information Systems* **40** (2013), no. 1 63–84.