Mapping Simple, Acyclic Rules to Linear Temporal Logic Formulas

ISAAC MACKEY, University of California, Santa Barbara
JIANWEN SU, University of California, Santa Barbara

A business service consists of a set of business processes. An enactment of a business service is formed from instances of the service’s constituent processes. Business service rules are conditions restricting enactments to comply with policies and regulations, and to honor service-level agreements with clients. The problem of service provisioning includes specifying business service rules, monitoring enactments to detect violations of rules at run-time, and maintaining and updating rules. In this paper, we formulate a technical model and techniques for service provisioning. Specifically, we develop a logic language for specifying rules and an approach towards automatically generating finite state machines (FSMs) as run-time monitors from a given set of rules. This approach involves two steps, translating: (1) rules to linear temporal logic (LTL) formulas, and (2) LTL formulas to FSMs. Since step (2) can be done by known algorithms, we focus on step (1), i.e., mapping rules to equivalent LTL formulas. For a subclass of rules called “simple” rules, we develop a technique to construct equivalent LTL formulas for each set of simple rules, and establish the correctness of this construction.

ACM Reference Format:

1 INTRODUCTION

Services make up a significant portion of the 21st century economy; in some countries, up to 80% of the Gross National Product (GNP) is attributed to services [1]. Service providers manage resources that provide services into service systems. When these service systems combine human managers and software infrastructure, service providers can apply principles from business process management (BPM), web services, and service-oriented computing paradigms. Furthermore, when a service is constrained by business rules, the actions of the service system, i.e. how the service is provisioned, must be controlled accordingly. In this paper, we initiate a study on “service provisioning” and investigate the use of automated verification techniques to benefit service providers.

A business process is a collection of related activities that accomplish a specific goal. Standards developed by the BPM and web services communities like Business Process Execution Language (BPEL), Web Service Description Language (WDSL), and Representational State Transfer (REST) framework, aim at using business processes in the design of (web service) software systems. These standards and other work in service composition (choreography and orchestration) focus on individual processes and/or composed processes. A service, however, typically consists of a group of related but not composed processes. While extensive research has been done on service compositions, there is less work on modeling temporal or data-centric relationships between processes, even though these relationships are equally important in

Authors’ addresses: Isaac Mackey, University of California, Santa Barbara, isaac_mackey@ucsb.edu; Jianwen Su, University of California, Santa Barbara, su@cs.ucsb.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.

Manuscript
determining service behavior and quality [30]. Modeling services with inter-process relationships, rather through explicit composition, gives service providers more flexibility in determining how enactments of services should progress.

To model business services, the notion of an “enterprise process framework” was formulated [30]. This framework consists of four core elements: a data model, a set of processes, a set of relationships, and a set of Key Performance Indicators (KPIs). While specification languages have been developed for data modeling [15], process modeling [13], and KPI formulation [22], modeling relationships between processes requires the development of new techniques. In practice, such relationships between processes are called “business rules”, “service rules”, etc., and define logical requirements (e.g., sanity checks), that reflect the service provider’s limitations, restrictions on the client’s freedom (e.g., constraints), or requirements derived from business goals and corporate policies (e.g., business rules). Ideally, all such relationships must be examined for or enforced on enactments of services.

Rules often originate in natural language, but need to be interpreted unambiguously and implemented in a programming language or formal specification language to be applied to a service system. While Semantics of Business Vocabulary and Rules [29] and, to a limited degree, DecSerFlow [31] may serve as specification languages, more study is needed to understand which requirements can be expressed in such languages and which cannot.

Once an service system is modeled in an enterprise process framework and service rules specified as process relationships, the challenge of service provisioning can be addressed, that is, the service provider must ensure that all enactments of the service system comply with the rules. In the last decade, this complianc has been achieved through formal verification. [5, 8]. Verification examines all possible process behaviors to ensure that no violations to specified (temporal) properties will occur. However, service provisioning is different from verification. Indeed, because the service’s component processes are not composed, violations of rules may arise from “unscripted”, e.g., ad hoc, behaviors by clients. Therefore, the challenge of upholding rules while providing services deserves further study.

Our approach to service provisioning is based on the general paradigm outlined in [30], as follows. We formulate a model for services (schema and enactments) as a set of business processes. Temporal patterns (constraints) for business processes in the same service enactment are given by conjunctive formulas with arithmetic inequalities over timestamps associated with process executions. Process relationships are represented as rules between these patterns. Based on this model, we envision an approach where a finite-state machine monitors the compliance of service enactments with respect to a set of process relationships. We use temporal logics as a bridge from service rules to finite-state machines; because existing results convert LTL to finite-state machines, leaving a gap between rules and LTL; thus we develop a technique to map rules to LTL formulas.

This paper makes the following technical contributions. We develop:

- A formal model for business services and rules, with a specification language to express quantitative temporal relationships between business processes,
- An approach to service provisioning based on finite-state machines,
- Translation techniques to map simple rules to (future and past) LTL formulas, with correctness of translation established.

The paper is organized as follows. Section 2 motivates service provisioning through an example service. Section 3 defines a technical model for service enactments, a language for service rules, and the linear temporal logic (LTL) with past and future operators used in this work. Section 4 presents a mapping from rules to LTL. Section 5 discusses related work. Finally, Section 6 concludes the paper.
2 MOTIVATIONS

This section illustrates the problem of service provisioning using an example Infrastructure-as-a-Service (IaaS) provider. We also describe the need for automated techniques for service provisioning at runtime and outline a solution using finite state machines.

Consider an IaaS provider that owns thousands of commodity servers and database machines and rents these to clients as a service named RentMe. The service is performed by business processes, some initiated by clients, others by the service provider. More specifically, the RentMe service includes the following business processes (in the typewriter font): A client may Request access by providing a description of their desired machine instances. A Schedule process performed by the service provider reserves a specific machine for a client. RentMe clients can Compute on assigned machines and Terminate instances. Also, a client makes a Payment for the computing services rendered and receive a Receipt from the service provider.

The set of process instances serving a client forms a (service) enactment. An enactment may contain tens or hundreds of process instances, as a client may initiate many requests, compute on many servers, and make many payments. Each process instance in an enactment is tagged with a unique instance id (iid) as well as a timestamp (e.g., a date for RentMe) for completion (or initiation) of the process instance. Processes instances also have other data; for example, each Schedule and Compute instance includes an identifier for the machine scheduled (resp., used). An enactment can be viewed as a set of relations, one for each process. Fig. 1 illustrates a partial enactment with 3 sets of process instances: For example, Row 1 in the Schedule table corresponds to a Schedule process instance with iid 101 completed on Jan. 3, 2020 that assigns machine 445 to the RentMe client.

<table>
<thead>
<tr>
<th>iid</th>
<th>#machines</th>
<th>timestamp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>5</td>
<td>2020-01-01</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>5</td>
<td>2020-01-06</td>
<td></td>
</tr>
<tr>
<td>002</td>
<td>10</td>
<td>2020-01-25</td>
<td></td>
</tr>
<tr>
<td>003</td>
<td>5</td>
<td>2020-01-30</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>iid</th>
<th>Request iid</th>
<th>machine</th>
<th>timestamp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>000</td>
<td>445</td>
<td>2020-01-03</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>000</td>
<td>446</td>
<td>2020-01-04</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>001</td>
<td>447</td>
<td>2020-01-08</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>001</td>
<td>447</td>
<td>2020-02-11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>iid</th>
<th>machine</th>
<th>timestamp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>445</td>
<td>2020-01-05</td>
<td></td>
</tr>
<tr>
<td>401</td>
<td>445</td>
<td>2020-01-18</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>iid</th>
<th>amount</th>
<th>timestamp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>$150</td>
<td>2020-01-15</td>
<td></td>
</tr>
<tr>
<td>501</td>
<td>$155</td>
<td>2020-01-30</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>iid</th>
<th>timestamp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>2020-01-07</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Process instances generated by one RentMe client

We consider when all RentMe enactments are subject to regulations or policies known to the service provider, including those specific to the RentMe service as specified in a service-level agreement (SLA). The collection of requirements set by service policies, SLAs, and regulations are usually known as business rules. For example, it may be required that valid requests are scheduled in a timely manner or an initial payment is made before a new client is granted user-level access to machines. The service provider should regularly access and inspect machines under its provenance to guard against rule violations. The service provisioning problem naturally rises here: how to ensure all service enactments are compliant with a set of rules?

Consider a RentMe rule named PromptSchedule, which states each client should Schedule their machines no later than three days after a Request process.
If the client fails to schedule their requested machines, their machines are reclaimed, thus the sooner a violation is detected, the sooner machines will be reclaimed and made usable for other clients. For this reason, we state one desirable property of a service provisioning algorithm:

*Provisioning Objective 1:* Rule violations should be detected promptly.

Furthermore, when a violation may be anticipated and prevented by action taken by the client or the service provider. Hence it is desirable to have:

*Provisioning Objective 2:* Rule violations should be proactively avoided.

In the last two decades, formal verification (e.g., model checking) has been applied to analyze services and service compositions. In static verification, elements of a service are mapped to a formal process model, e.g. a Petri net [11] or finite state machines [4, 19], and *every possible* execution sequence of this model is checked for satisfaction with respect to some desirable properties or constraints, often written in a logic specification language, e.g. LTL. Unfortunately, static verification cannot be applied when some participants in a service execution do not have prescribed behaviors to follow. For example, a *RentMe* client can initiate some process (e.g., *Request*, *Compute*, *Payment*) at any point during the *RentMe* service at their discretion. Including such “arbitrary” behaviors by these participants may lead to failure of compliance checking. Thus the infeasibility of prevention rule violations with static verification failure does not indicate a design flaw, rather the flexibility of our service model.

Because static verification is infeasible, service providers often address the provisioning problem in practice with *runtime monitoring*. The prevailing form of runtime monitoring is to embed business rules in the target application’s software [3, 14]. There are several disadvantages to this approach. First, changes to rules must be re-engineered into software code; the cost of updating rules includes costly software development. Second, the overhead of checking rules may increase the time and space usage of an application. Third, programmatic guards may indicate if an enactment satisfies or violates a rule, but they does not provide an efficient representation of how an enactment can be safely advanced, i.e. they do not address the second provisioning objective.

In this paper, we approach service provisioning using a logical monitor, a mechanism separated from the application that observes its execution trace. This separation eases the management of service rules by abstracting system-level details out of the rule specification and monitoring mechanisms. Often, a logical monitor is a finite state machine generated from specifications of rules or consists of algorithms for incrementally evaluating rule satisfaction with respect to data from the execution trace e.g. [24]. The benefits of this approach to service providers are many. By monitoring rules automatically, a source of human error is removed, leading to increased confidence that service enactments adhere to business goals and policies. Also, only the history relevant to a rule’s satisfaction is recorded by a runtime monitor; this approach naturally optimizes parts of the cost of formal verification. Third, separating the implementation of the service from the monitoring mechanism means changes to rules can be implemented quickly.

*Example 2.1.* To illustrate this idea, consider the rule *PromptSchedule* in *RentMe*: *For each client, if a Request instance completes, a Schedule instance must complete on that same day or in the next three days.* This rule can be written in a logic language as:

\[
\text{Request}@x \rightarrow \text{Schedule}@y, x \leq 0 \land y \geq 3
\]

We monitor this rule for a single *RentMe* enactment using the following finite state machine (Figure 2). The machine transitions once per day, changing state based on the process instances observed for the previous 24 hours, so its alphabet consists of subsets of \{R, S\} (R for *Request*, S for *Schedule*). The start state, state 1, is accepting because initially
no request requiring a subsequent schedule instance has been observed. When a request takes place with a matching schedule process, the rule PromptSchedule remains satisfied and the machine remains in state 1. The machine moves from state 1 to state 2 when request takes place without a simultaneous schedule process. 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 process instance. If an machine is in state 4 and the following day fails to contain a schedule process, the machine enters the sink state 5, as the enactment has invariably violated PromptSchedule. If a Schedule process is completed when the machine is in state 2, 3, or 4, an unscheduled request has been matched and the machine returns to state 1, which is accepting. In addition to using this machine for detecting rule violations, we can use this machine to prevent rule violations. If the machine is in state 4 a process manager can prompt the client to schedule their requested machines immediately.

As the previous example illustrates, we can translate rules to finite state machines and use these machines to guide service provisioning. In the technical development, we focus on translating rules to LTL formulas. LTL formulas can then be converted into finite state machines using known translations [20, 32]. Accordingly, if all service rules can be translated into temporal logic, then into finite state machines, dynamically provisioning services requires simply transitioning states of a finite state machine and checking the reachability of accepting states—a much simplified task. To provision services with respect to a set of rules, one simply constructs a product machine. Summarizing the above discussions, the central technical problem in our approach to service provisioning is the following rule translation problem: Given a rule over a service, construct an equivalent LTL formula.

3 A MODEL FOR SERVICE PROVISIONING

In this section, we introduce key technical notions used in this paper. First, we define a “service schema” and an “enactment” as the model of services. We then define “constraints” to describe temporal relationships between process instances and “service rules” to describe relationships between pairs of constraints. The section concludes with presenting a (past- and future-time) linear temporal logic needed for stating the main result of the paper (Sec. 4).

To begin, we assume the existence of three countably infinite, pairwise disjoint sets:

- $\mathcal{P}$ of process names,
• \( \mathbb{N} \) of natural numbers, to serve as a set of discrete, ordered timestamps, along with the integers \( \mathbb{Z} \supset \mathbb{N} \) for technical development, and
• \( \mathcal{V} \) of variables (to hold timestamps).

The set of processes involved in a service constitutes a schema for that service. Given a schema, a collection of instances for each process name in the schema constitutes an enactment of that service. We focus on temporal properties of enactments, so we associate each process instance in an enactment with a timestamp, with 1 as the smallest timestamp. Furthermore, we assume that for each process, different instances have distinct timestamps. Thus, an enactment of a service schema can be faithfully represented by a set of unary relations over the domain of timestamps \( \mathbb{N} - \{0\} \).

**Definition:** A service schema is a finite subset of \( \mathcal{P} \). An enactment \( \eta \) of a service schema \( S \) is a mapping \( \eta: S \rightarrow \mathbb{N}^{\mathbb{N} - \{0\}} \) such that \( \eta(p) \) is finite for all \( p \in S \).

**Example 3.1.** For the RentMe service discussed in Sec. 2, the set of processes \( S_{\text{RM}} = \{\text{Request}, \text{Schedule}, \text{Compute}, \text{Terminate}, \text{Payment}, \text{Receipt}\} \) forms a service schema. For the process instances of RentMe, we use the number of days into the year 2020 as the natural number timestamp for a given instance. For example, the date 2020-01-04 is given the timestamp 4, Fig. 1, shows an enactment \( \eta \) of the service schema can be faithfully represented by a set of unary relations over the domain of timestamps \( \mathbb{N} - \{0\} \).

We now define the syntax of our rule language. We start with two types of atomic formulas, “process atoms” and “gap atoms”. For each \( n \in \mathbb{Z} \) we use \( \leq_n \) and \( \geq_n \) as binary predicates, on the domain of timestamps.

**Definition:** Let \( n \in \mathbb{Z} \) we use \( \leq_n \) and \( \geq_n \) as binary predicates, on the domain of timestamps.

A process atom \( p@x \) expresses that an instance of process \( p \) happens at time \( x \), e.g. \( \text{Request}@x \) denotes an instance of the Request process at time \( x \). An atom \( x \leq_n y \) means \( x + n \leq y \), expressing that the time \( x + n \) no later than time \( y \). When \( n > 0 \), \( x \leq_n y \) indicates a delay from time \( x \) to time \( y \) of at least \( n \) time units. Similarly, an atom \( x \geq_n y \) means \( x + n \geq y \), expressing that the time \( x + n \) is no earlier than time \( y \). When \( n > 0 \), \( x \geq_n y \) indicates a bound on time \( y \) following time \( x \): \( y \) is at most \( n \) time units after \( x \) and potentially simultaneous with or before \( x \). Note that for each positive \( n \in \mathbb{N} \), \( x \geq_n y \) does not imply \( x \geq y \), as in \( 4 \geq_2 5 \).

**Definition:** Let \( S \) be a service schema. A constraint over \( S \) is a finite set of process atoms over \( S \) and gap atoms. A constraint \( \phi \) over \( S \) is closed if each variable used in \( \phi \) occurs in some process atom in \( \phi \). Let \( \text{var}(\phi) \) and \( \text{proc}(\phi) \) denote the set of variables and the set of process names used in \( \phi \), resp.

**Example 3.2.** The constraint \( \{\text{Request}@x, \text{Schedule}@y, x \leq_4 y, x \geq_6 y\} \) is closed over \( S_{\text{RM}} \). Intuitively, this constraint selects a Request instance and a Schedule instance that follows the Request by exactly four, five, or six days. The constraint \( \{\text{Request}@x, \text{Schedule}@y, z \geq_5 y\} \) over \( S_{\text{RM}} \) is not closed, and it selects a pair of Request and Schedule instances where the Schedule instance occurs no more than five days after some time \( z \).
Finally, we define a syntax for “rules” to express relationships between constraints over the same service schema. The intent of a rule is to require that each set of process instances selected by one constraint can be extended to satisfy another constraint.

**Definition:** Let \( S \) be a service schema. A (service) rule over \( S \) is an expression \( \phi \rightarrow \psi \) where \( \phi, \psi \) are constraints over \( S \) such that both \( \phi \) and \( \phi \cup \psi \) are closed.

Note that \( \psi \) is not required to be closed. We now briefly describe the semantics of rules. The satisfaction of a constraint \( \phi \) over \( S \) is defined with respect to an enactment \( \eta \) of \( S \) and an assignment, i.e., a mapping \( \sigma : V \rightarrow \mathbb{N} \). We say \( \eta \) satisfies \( \phi \) with \( \sigma \), denoted \( \eta \models \phi[\sigma] \), if (1) for each process atom \( p \circ \pi \) in \( \phi \), \( \sigma(\pi) \in \eta(p) \), and (2) for each gap atom \( x \leq y \) (resp. \( x \geq y \)) in \( \phi \), \( \sigma(x) \leq \sigma(y) \) (resp. \( \sigma(x) \geq \sigma(y) \)). Observe that a constraint is understood as a conjunction over a set of atoms. Two constraints are equivalent if they are satisfied by the same set of assignments for all enactments.

**Example 3.3.** Consider the enactment \( \eta \) in Fig. 1, and the constraint \( \phi = \{\text{Request} @ x, \text{Schedule} @ y, x < 0, y \geq 3 \} \) (a Request is Scheduled between 0 and 3 days after it is received), and the following two assignments \( \sigma_1(x \mapsto 1, y \mapsto 3) \) and \( \sigma_2(x \mapsto 1, y \mapsto 6) \). Then \( \eta \not\models \phi[\sigma_1] \) because 1 \( \not\in \eta(\text{Request}) \), 3 \( \in \eta(\text{Schedule}) \), 1 \( \leq 3 \), and 1 \( \geq 3 \). However, \( \eta \not\models \phi[\sigma_2] \) because 1 \( \not\geq 3 \).

Let \( S \) be a service schema, \( \eta \) an enactment of \( S \), and \( \phi \rightarrow \psi \) a rule over \( S \). Then \( \eta \) satisfies \( \phi \rightarrow \psi \), denoted as \( \eta \models \phi \rightarrow \psi \), if for each assignment \( \sigma \) such that \( \eta \models \phi[\sigma] \), there is an assignment \( \sigma' \) such that \( \eta \models \psi[\sigma'] \) and \( \sigma, \sigma' \) agree on \( \text{var}(\phi) \). For a set \( R \) of rules over \( S \), \( \eta \) satisfies \( R \), denoted \( \eta \models R \), if \( \eta \models r \) for each rule \( r \in R \).

**Example 3.4.** The Timely Payment rule requires each pair of Request and subsequent Schedule processes in an enactment to be followed within 3 days by a Payment process. This can be written as a rule:

\[
\text{rTimely Payment} : \{\text{Request} @ x, \text{Schedule} @ y, x < 0, y \geq 3\} \rightarrow \{\text{Payment} @ z, y < 0, z \geq 3\}
\]

The enactment \( \eta \) in Example 3.1 does not satisfy \( \text{rTimely Payment} \): For the Request instance at time 1 and the Schedule instance at time 3, there is no Payment instance with timestamp between 3 and 6.

Given a service schema \( S \) and a set of rules \( R \) over \( S \), Sec. 2 suggests determining if an enactment of \( S \) satisfies \( R \), i.e., solving the service provisioning problem, using finite state machines derived from linear temporal logic (LTL). We now define the LTL formulas over \( S \) used for our translation. Let \( S \) be a service schema; we treat each \( p \in S \) as a propositional variable. LTL formulas over \( S \) are defined recursively as follows:

\[
\varphi := p \mid \text{true} \mid \text{false} \mid \neg \varphi \mid \varphi \land \varphi \mid X \varphi \mid X^{-1} \varphi \mid F \varphi \mid P \varphi
\]

where \( p \in S \), true, false are Boolean constants, and \( \neg, \land \) are Boolean operators. The standard Boolean abbreviations \( \lor \) (or) and \( \rightarrow \) (implies) are used as well. The temporal operators \( X \text{ (next)} \) and \( F \text{ (future)} \) are common in future-time LTL [28], while \( P \text{ (past)} \) and \( X^{-1} \text{ (yesterday)} \) (sometimes written as \( Y \ [6] \)) are common in past-time LTL [21]. 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 formula are satisfied by “traces”, defined as follows. An interpretation is a mapping from a set of propositions \( S \) to \{true, false\}. A trace is a (finite) sequence of interpretations, such that for a trace \( \pi \), \( \pi \) has length \( \text{len}(\pi) = n \), and for \( 0 \leq i \leq \text{len}(\pi) - 1 \), \( i \) is a (time) instant in \( \pi \), and \( \pi[i] \) denotes the \( i^{th} \) interpretation in \( \pi \). For a trace \( \pi \), an instant \( i \) such that \( 0 \leq i \leq \text{len}(\pi) - 1 \), and an LTL formula \( \varphi \), we say \( \pi \) satisfies \( \varphi \) at instant \( i \), denoted \( \pi, i \models \varphi \), if one of the following is true (the cases for Boolean constants and operators are standard and thus omitted):
\begin{itemize}
  \item $\pi, i \models p$ if $\pi[i](p) = \text{true}$,
  \item $\pi, i \models X\varphi$ if $i < \text{len}(\pi) - 1$ and $\pi, i + 1 \models \varphi$,
  \item $\pi, i \models X^{-1}\varphi$ if $i > 0$ and $\pi, i - 1 \models \varphi$,
  \item $\pi, i \models F\varphi$ if $\exists j, i \leq j \leq \text{len}(\pi)$ and $\pi, j \models \varphi$, and
  \item $\pi, i \models P\varphi$ if $\exists j, 0 \leq j \leq i$ and $\pi, j \models \varphi$
\end{itemize}

Intuitively, the semantics indicate that $\pi$ satisfies $X^k\varphi$ (or $X^{-k}\varphi$) at an instant if $\varphi$ is satisfied at the $k^{th}$ following (resp. preceding) instant, and $\pi$ satisfies $F\varphi$ (or $P\varphi$) at an instant if $\varphi$ is satisfied at the current or some upcoming (resp. previous) instant.

Enactments are closely related to finite traces. Given an enactment $\eta$, let $\kappa$ be the largest timestamp present in $\eta$ and 0 if $\eta$ is empty. The following mapping connects enactments and traces.

Definition 3.5. Let $\eta$ be an enactment of a service schema $S$. The trace $\pi_\eta$ is the sequence $\pi_\eta[0] \cdots \pi_\eta[\kappa]$ where for each $i \in [0..\kappa]$ and each $p \in S$, $\pi_\eta[i](p) = \text{true}$ if $i \in \eta(p)$, and false otherwise. Conversely, for each trace $\pi$, the enactment $\eta_\pi$ is defined as follows: for each $i \in [0..\text{len}(\pi)]$ and each $p \in S$, $i \in \eta_\pi(p)$ if $\pi[i](p) = \text{true}$.

Below is the initial segment of the trace of the enactment derived from the set of process instances shown in Sec. 2, Fig. 1.

<table>
<thead>
<tr>
<th>$\pi_\eta[i]$</th>
<th>Request</th>
<th>Schedule</th>
<th>Schedule</th>
<th>Compute</th>
<th>Request</th>
<th>Terminate</th>
<th>Schedule</th>
<th>...</th>
<th>Compute</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The main technical problem studied in this paper can now be stated: Given a set of rules $R$ over a service schema $S$, is there an LTL formula $\varphi$ such that for each enactment $\eta$ of $S$, $\eta \models R$ iff $\pi_\eta, 1 \models \varphi$?

4 MAPPING RULES TO LTL

This section introduces a subclass of rules called "simple rules" and presents the main technical result of this paper: a translation from simple rules to LTL.

First, we introduce a mechanism for mapping gap atoms to LTL operators. Then, we introduce a graph representation of constraints used in a translation function for connected, acyclic constraints. Then, we state and prove a lemma concerning the correctness of this translation. Then, we provide a function to translate arbitrary acyclic constraints. Finally, we define simple rules and a translation function for simple rules using our translation function for acyclic constraints as a subroutine.

Example 4.1. Consider the constraint $\phi_{\text{rental}}$:

\[ [\text{Request}@x, \text{Schedule}@y, \text{Compute}@z, \text{Terminate}@w, x \leq_1 y, x \geq_1 10 y, y \geq_5 z, z \leq_0 w] \]

Intuitively, $\phi_{\text{rental}}$ selects timestamps $x, y, z, w$ for instances of processes Request, Schedule, Compute, Terminate resp. that satisfy the following three conditions:

(i) Schedule occurs at least one day after but no more than 10 days after Request, i.e. Request occurs at $x$ and Schedule occurs at $y$ with $x + 1 \leq y \leq x + 10$,

(ii) Compute occurs no later than 5 days after Schedule, i.e. Schedule occurs at $y$ and Compute occurs at $z$ with $y + 5 \geq z$, and

(iii) Compute occurs before or simultaneously with Terminate, i.e. Compute occurs at $z$ and Terminate occurs at $w$ with $z \leq_0 w$. 

Manuscript
Condition (i) is observed in a trace if the proposition \( \text{Request} \) holds at instant \( x \) and the proposition \( \text{Schedule} \) holds at instants \( x + 1, x + 2, \ldots, \) or \( x + 10 \). We rewrite the second statement using next LTL operators: \( \text{Schedule}, \text{X}^2 \text{Schedule}, \ldots \) or \( \text{X}^{10} \text{Schedule} \) holds at \( x \). Using boolean LTL operators produces a translation of (i): \( \text{Request} \land \bigvee_{1 \leq j \leq 10} \text{X}^j \text{Schedule} \).

Condition (ii) is observed in a trace if \( \text{Schedule} \) holds at \( y \) and \( \text{Compute} \) holds at or before \( y + 5 \). We rewrite the second statement with next and past LTL operators: \( \text{X}^5 \text{P Compute} \) holds at \( y \). Thus, our translation of condition (ii) is: \( \text{Schedule} \land \text{X}^5 \text{P Compute} \).

Condition (iii) is observed in a trace if \( \text{Compute} \) holds at \( z \) and \( \text{Terminate} \) holds at \( z \) or a later instant. Rewriting the second statement with the future LTL operator, we have condition (iii) when \( \text{Compute} \) and \( \text{F Terminate} \) hold at \( z \). Thus, our translation of condition (iii) is \( \text{Compute} \land \text{F Terminate} \).

We generalize these examples with the following function.

**Definition:** Given a set of gap atoms \( \phi \) over two variables \( x \) and \( y \), let the function \( e_x(\phi) \) be the following operator combinations:

\[
    e_x(\phi) = \begin{cases} 
        \text{X}^n \text{F} & \text{if } \phi \text{ is equivalent to } \{ x \leq_n y \text{ for some } n \in \mathbb{Z} \} \\
        \text{X}^m \text{P} & \text{if } \phi \text{ is equivalent to } \{ x \geq_m y \text{ for some } m \in \mathbb{Z} \} \\
        \bigvee_{n \leq j \leq m} \text{X}^j & \text{if } \phi \text{ is equivalent to } \{ x \leq_n y, x \geq_m y \text{ for some } n, m \in \mathbb{Z} \}
    \end{cases}
\]

Note that this function generates the translations shown in Example 4.1. For example, for condition (i), \( \text{Request@x} \) becomes the proposition \( \text{Request} \). Applying \( e_x \) to the gap atoms \( \{ x \leq_1 y, x \geq 10 y \} \) produces the operators \( \bigvee_{1 \leq j \leq 10} \text{X}^j \), and the proposition corresponding to variable \( y \), \( \text{Schedule} \), is placed after these operators, yielding \( \text{Request} \land \bigvee_{1 \leq j \leq 10} \text{X}^j \text{Schedule} \).

Next, we use properties of inequality predicates to show that all constraints using two variables can be translated with the \( e \) function. More specifically, we identify cases where some gap atoms are redundant, i.e. implied by other gap atoms.

**Lemma 4.2.** Let \( \phi \) be a set of gap atoms where \( \text{var}(\phi) = \{ x, y \} \). Then \( \phi \) is equivalent to one of the following three sets:

- \( \{ x \leq_m y \} \) for some \( n \in \mathbb{Z} \)
- \( \{ x \geq_m y \} \) for some \( m \in \mathbb{Z} \)
- \( \{ x \leq_m y, x \geq_m y \} \) for some \( n, m \in \mathbb{Z} \)

**Proof.** First, observe that for all \( n, m \in \mathbb{Z} \), \( y \leq_m x \) iff \( x \geq_{-m} y \), and \( y \geq_m x \) iff \( x \leq_{-m} y \). These facts allow all gap atoms in \( \phi \) to be written with \( x \) on the left of the predicate and \( y \) on the right.

Second, observe that for all \( n, m \in \mathbb{Z} \) with \( n \leq m \), \( x \leq_m y \) implies \( x \leq_n y \), and \( x \geq_n y \) implies \( x \geq_m y \). If \( \phi \) contains \( x \leq_n y \) and \( x \leq_m y \) where \( n \leq m \), then \( \phi \) is equivalent to \( \phi - \{ x \leq_n y \} \). A similar statement holds for \( \geq \)-atoms. Observations like these can be repeated to identify a subset of \( \phi \) with at most one \( \leq \)-atom and one \( \geq \)-atom equivalent to \( \phi \).

**Example 4.3.** Let \( \phi_4 = \{ x \leq_1 y, x \leq_3 y, x \geq 4 y, y \geq 1 x \} \). First, we rewrite all gap atoms with \( x \) on the left of the predicate: \( \phi_4 = \{ x \leq_1 y, x \leq_3 y, x \geq 4 y, x \leq_{-1} y \} \). Note that \( x \leq_3 y \) implies \( x \leq_1 y \) and \( x \leq_3 y \) implies \( x \leq_{-1} y \), so \( x \leq_1 y \) and \( y \geq_1 x \) can be removed from \( \phi_4 \). Thus, \( \phi_4 \) is equivalent to \( \{ x \leq_3 y, x \geq 4 y \} \).
Now, we translate a constraint with more than two variables. To do this, we introduce a technique for “joining” two LTL formulas of constraints that share one variable. We illustrate this technique with the following example.

**Example 4.4.** Consider \( \phi_{\text{rental}} \) from Example 4.1 and its decomposition into three conditions. Condition (ii) covers atoms \{Schedule@y, Compute@z, y \geq 5 z\} and can be translated as:

\[
\text{Schedule} \land X^5 P \text{Compute}
\]

Condition (iii) covers atoms \{Compute@z, Terminate@w, z \leq 0 w\} and can be translated as:

\[
\text{Compute} \land F \text{Terminate}
\]

These two sets of atoms share the variable \( z \), which corresponds to the \text{Compute} proposition in both LTL translations. Accordingly, we can combine the translations by placing the second formula in the position in the first formula where \text{Compute} appears, combining the duplicated proposition (bars added for illustration):

\[
\psi = \text{Schedule} \land X^5 P (\text{Compute} \land F \text{Terminate})
\]

To complete the translation of \( \phi_{\text{rental}} \), we join \( \psi \), with our LTL translation of condition (i). Recall that condition (i) covers atoms \{Request@x, Schedule@y, x \leq 1 y, x \geq 10 y\} and can be translated as

\[
\text{Request} \land \bigvee_{1 \leq j \leq 10} X^j \text{Schedule}
\]

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

\[
\phi_{\text{rental}}^{\text{LTL}} = \text{Request} \land \bigvee_{1 \leq j \leq 10} X^j (\text{Schedule} \land X^5 P (\text{Compute} \land F \text{Terminate}))
\]

In the example below, we demonstrate the satisfaction of \( \phi_{\text{rental}}^{\text{LTL}} \) from Example 4.1 and the corresponding satisfying assignment for \( \phi_{\text{rental}} \).

To generalize the joining technique in Example 4.4, we use a graph representation of constraints. Recall that each constraint is a conjunctive formula with process atoms (unary predicates) and gap atoms (binary predicates) over a set of variables. These constraint have a faithful representation as an undirected graph.

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

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

Furthermore, \( \phi \) is *acyclic* if \( G_\phi \) is acyclic, *connected* if \( G_\phi \) is connected.

**Example 4.5.** The graph of \( \phi_{\text{rental}} \) (Fig. 3) has nodes \( \{x, y, z, w\} \) labeled with Request, Schedule, Compute, Terminate resp., edges \((x, y), (y, z), (z, w)\) labeled with \{\[x \leq 1 y, x \geq 10 y\], \[y \geq 5 z\], \[z \leq 0 w\]\} resp. and is acyclic and connected.
Definition: Let $S$ be a service schema and $\phi$ an acyclic, connected constraint over $S$, such that $G_\phi = (V, E, L)$ is the graph of $\phi$. The derived tree of $\phi$ at $x$, denoted $T_\phi^x$ is the directed tree 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 $z \in V$, let $Ch(z)$ denote $z$’s children.

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

$$AcycConn(T_\phi^x, y) = \begin{cases} \bigwedge_{p \in L(y)} p & \text{if } y \text{ is a leaf} \\ \bigwedge_{p \in L(y)} p \land \bigvee_{z \in Ch(y)} e_y(L(y, z)) \text{ AcycConn}(T_\phi^z, z) & \text{otherwise} \end{cases}$$

The process names that label $y$ are used in a conjunction of LTL propositions, i.e. $\bigwedge_{p \in L(y)} p$. Then, for each child $z$ of $y$, the gap atoms $L(y, z)$ are translated to to the LTL operators $e_y(L(y, z))$ and the algorithm makes a recursive call AcycConn($T_\phi^z$, $z$) to translate the subtree rooted at $z$ with respect to $z$.

Example 4.6. The AcycConn translation of $\phi_{\text{Rental}}$ abbreviated here as $\phi$, is done using the derived tree $T_\phi^x$ shown in Fig. 4. Let $\psi_y, \psi_z, \psi_w$ be the subsets of $\phi$ such that the derived trees $T_\phi^{\psi_y}, T_\phi^{\psi_z}, T_\phi^{\psi_w}$ are the subtrees of $T_\phi^x$ rooted at $y, z, w$ (resp.). We demonstrate the translation beginning with the leaf node $w$ and towards the root $x$:

$$AcycConn(T_\phi^{w}) = \text{Terminate}$$
$$AcycConn(T_\phi^{z}) = \text{Schedule} \land T_\phi^{w} \text{AcycConn}(T_\phi^{w}) = \text{Schedule} \land \text{F Terminate}$$
$$AcycConn(T_\phi^{y}) = \text{Compute} \land T_\phi^{z} \text{AcycConn}(T_\phi^{z}) = \text{Compute} \land T_\phi^{w} \text{AcycConn}(T_\phi^{w}) \bigwedge \text{X}^2 \text{P (Compute } \land \text{F Terminate) }$$
$$AcycConn(T_\phi^{x}) = \text{Request} \land \left( \bigvee_{-10 \leq j \leq -1} \text{Schedule} \land T_\phi^{y} \text{AcycConn}(T_\phi^{y}) \bigwedge \text{X}^2 \text{P (Compute } \land \text{F Terminate) } \right)$$

The following lemma relates two notions of satisfaction defined in Section 3: 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 $\phi$ be an acyclic, connected constraint where $x, y \in \text{var}(\phi)$. We use $\text{atoms}(T_\phi^x)$ and $\text{atoms}(T_\phi^y)$ to denote the set of atoms $\sigma$ in $\phi$ such that each variable in $\sigma$ is a node in $T_\phi^x$ (resp. $T_\phi^y$).

Lemma 4.7. Let $S$ be a service schema, $\eta$ an enactment of $S$, $\phi$ a connected, acyclic constraint over $S$, $x, y$ arbitrary variables in $\text{var}(\phi)$, where $T_\phi^x = (V, E, L)$ is a derived tree of $\phi$ rooted at $x$ and $T_\phi^y$ the subtree of $T_\phi^x$ rooted at $y$. For each timestamp $i$, the following statements are equivalent:

- For some assignment $\sigma$, $\sigma(y) = i$ and $\eta \models \text{atoms}(T_\phi^y)[\sigma]$
\[ \pi_{\eta, i} \models \text{AcycConn}(T^x_{\phi}, y) \]

**Proof.** Lemma 4.7 is shown by induction on the height of \( y \) in the tree \( T^x_{\phi} \), equivalently, on the height of the subtree \( T^x_{\phi} \).

Let \( i \) be an arbitrary timestamp.

In the basis case, \( y \) is a leaf in \( T^x_{\phi} \). Because \( T^x_{\phi} \) is connected, \( T^x_{\phi} \mid y \) has just one node, \( y \). The atoms for \( T^x_{\phi} \mid y \) are the process atoms in \( \phi \) that use variable \( y \). Recall that \( L \) is the label function for \( T^x_{\phi} \). Then \( \text{atoms}(T^x_{\phi} \mid y) = \{ p@y \mid p \in L(y) \} \) and the \( \text{AcycConn} \) translation of \( T^x_{\phi} \) with respect to \( y \) is: \( \bigwedge_{p \in L(y)} p \). To prove the basis case, we show that for some assignment \( \sigma \), \( \sigma(y) = i \) and \( \eta \models \{ p@y \mid p \in L(y) \}[\sigma] \) if and only if \( \pi_{\eta, i} \models \bigwedge_{p \in L(y)} p \).

First, we assume the first statement and derive the second: Assume for some assignment \( \sigma \), \( \sigma(y) = i \) and \( \eta \models \{ p@y \mid p \in L(y) \}[\sigma] \). Writing the satisfaction of the process name propositions in a conjunction, we have: \( \pi_{\eta, i} \models \bigwedge_{p \in L(y)} p \).

Next, we assume the second statement and derive the first: Assume \( \pi_{\eta, i} \models \bigwedge_{p \in L(y)} p \). Isolating the satisfaction of each process atom, we have: for all process names \( p \) in the label \( L(y) \), \( \sigma(y) \in \eta(p) \). Using with the equality \( \sigma(y) = i \) yields: for all \( p \in L(y) \), \( i \in \eta(p) \). Under the mapping between enactments and traces given in Definition 3.5, this is equivalent to the statement: for all \( p \in L(y) \), \( \pi_{\eta, i} \models p \). Writing the satisfaction of the process name propositions in a conjunction, we have: \( \pi_{\eta, i} \models \bigwedge_{p \in L(y)} p \).

The inductive hypothesis states that for each \( z \in \text{Ch}(y) \), for each timestamp \( j \), the following statements are equivalent:

- For some assignment \( \sigma \), \( \sigma(z) = j \) and \( \eta \models \text{atoms}(T^x_{\phi} \mid z)[\sigma] \)
- \( \pi_{\eta, j} \models \text{AcycConn}(T^x_{\phi}, z) \)

For the inductive step, we show that for some assignment \( \sigma \), \( \sigma(y) = i \) and \( \eta \models \text{atoms}(T^x_{\phi} \mid y)[\sigma] \), if and only if \( \pi_{\eta, i} \models \text{atoms}(T^x_{\phi} \mid y)[\sigma] \) and \( \pi_{\eta, j} \models \text{AcycConn}(T^x_{\phi}, z) \).

First, we assume the first statement and derive the second: Assume for some assignment \( \sigma \), \( \sigma(y) = i \) and \( \eta \models \text{atoms}(T^x_{\phi} \mid y)[\sigma] \). Let \( z \) be an arbitrary child of \( y \), where \( T^x_{\phi} \mid z \) is the subtree rooted at \( z \). Because \( T^x_{\phi} \mid z \) is a subtree of \( T^x_{\phi} \) and \( \eta \models \text{atoms}(T^x_{\phi} \mid y)[\sigma] \), we have \( \eta \models \text{atoms}(T^x_{\phi} \mid z)[\sigma] \). Let \( j = \sigma(z) \). We apply the inductive hypothesis with \( \sigma \), \( j \), and \( T^x_{\phi} \mid z \), which yields \( \pi_{\eta, j} \models \text{AcycConn}(T^x_{\phi}, z) \).

The label \( L(y, z) \) is a constraint that restricts the gap between \( y \) and \( z \). Because \( \eta \models \text{atoms}(T^x_{\phi} \mid y)[\sigma] \) and \( L(y, z) \) labels the edge between \( y \) and a descendant \( z \), we have \( \eta \models L(y, z)[\sigma] \). Then, each assignment that maps \( y \) to \( z \) to \( j \) satisfies \( L(y, z) \).

Claim: Given that each assignment that maps \( y \) to \( i \) and \( z \) to \( j \) satisfies \( L(y, z) \) and \( \pi_{\eta, j} \models \text{AcycConn}(T^x_{\phi}, z) \), then \( \pi_{\eta, i} \models e_y(L(y, z)) \text{AcycConn}(T^x_{\phi}, z) \).

Case 1: \( L(y, z) \equiv \{ y \leq_{m} z \} \) for some \( m \in \mathbb{Z} \). Because \( \{ y \mapsto i, z \mapsto j \} \) satisfies \( \{ y \leq_{m} z \} \), we have \( i + m + k = j \) for some \( k \in \mathbb{N} \). Rewriting the result of the inductive hypothesis with this equality yields \( \pi_{\eta, i} \models e_y(L(y, z)) \text{AcycConn}(T^x_{\phi}, z) \).

Case 2: \( L(y, z) \equiv \{ y \geq_{m} z \} \) for some \( m \in \mathbb{Z} \). Because \( \{ y \mapsto i, z \mapsto j \} \) satisfies \( \{ y \geq_{m} z \} \), we have \( i + m - k = j \) for some \( k \in \mathbb{N} \). Rewriting the result of the inductive hypothesis with this equality yields \( \pi_{\eta, i} \models e_y(L(y, z)) \text{AcycConn}(T^x_{\phi}, z) \)
\( \neg k \) can be replaced by the past operator \( P \), then \( m \) can be replaced by \( m \) next (or \( m \) yesterday, if \( m \leq 0 \)) operators, yielding \( \pi, i \models X^mP \text{AcycConn}(T^n_\phi, z) \).

Case 3: \( L(y, z) \equiv [y \leq_m z, y \geq_m z] \) for some \( m, m' \in \mathbb{Z} \). Because \( [y \mapsto i, z \mapsto j] \) satisfies \( [y \leq_m z, y \geq_m z] \), we have \( i + k = j \) for some \( i < k \leq m \). We rewrite the result of the inductive hypothesis with this equality: \( \pi, i + k \models \text{AcycConn}(T^n_\phi, z) \) for some \( m \leq k \leq m' \). We rewrite this statement using the next (or yesterday, if \( m \leq 0 \)) operators: \( \pi, i \models X^k \text{AcycConn}(T^n_\phi, z) \). Then, we expand this as a disjunction over a range of values including \( k \): \( \pi, i \models \bigwedge_{m \leq k \leq m'} X^k \text{AcycConn}(T^n_\phi, z) \).

By Lemma 4.2, these cases for \( L(y, z) \) are exhaustive. Then, for each child \( z \) of \( y \), \( \pi, i \models e_y(L(y, z)) \text{AcycConn}(T^n_\phi, z) \).

Aggregating the claim for each child yields:
\[
\pi, i \models \bigwedge_{\eta \in \text{Ch}(y)} e_y(L(y, z)) \text{AcycConn}(T^n_\phi, z).
\]

Finally, the statement \( \eta \models \text{atoms}(T^n_\phi) [\eta] \) indicates that for each process atom \( \alpha \) associated with the root \( y \), \( \eta \models \alpha[\eta] \). This means for all process names \( p \in L(y) \), \( \sigma(y) \in \eta(p) \). Rewriting the second term with the equality \( \sigma(y) = i \) yields for all \( p \in L(y) \), \( i \in \eta(p) \). Under the mapping between enactments and traces, we have that for all \( p \in L(y) \), \( \pi, i \models p \). Equivalently, \( \pi, i \models \bigwedge_{p \in L(y)} p \). Combining this with the results of the claim, we have \( \pi, i \models (\bigwedge_{p \in L(y)} p) \land \bigwedge_{\eta \in \text{Ch}(y)} e_y(L(y, z)) \text{AcycConn}(T^n_\phi, z) \), i.e. \( \pi, i \models \text{AcycConn}(T^n_\phi, y) \).

Now we show the other direction of implication.

Assume \( \pi, i \models (\bigwedge_{p \in L(y)} p) \land \bigwedge_{\eta \in \text{Ch}(y)} e_y(L(y, z)) \text{AcycConn}(T^n_\phi, z) \). Let \( z \) be an arbitrary child of \( y \); we have \( \pi, i \models e_y(L(y, z)) \text{AcycConn}(T^n_\phi, z) \).

**Claim:** Given \( \pi, i \models e_y(L(y, z)) \text{AcycConn}(T^n_\phi, z) \), there is some timestamp \( j \) such that each assignment that maps \( y \) to \( i \) and \( z \) to \( j \) satisfies \( L(y, z) \) and \( \pi, j \models \text{AcycConn}(T^n_\phi, z) \). The proof is done by cases of \( L(y, z) \).

**Case 1:** \( L(y, z) \equiv [y \leq_m z] \). Rewriting the \( e_y \) function for this case, we have \( \pi, i \models X^mF \text{AcycConn}(T^n_\phi, z) \). The semantics of LTL allow us to move the index of satisfaction while removing next (or yesterday, if \( m \leq 0 \)) operators: \( \pi, i + m \models F \text{AcycConn}(T^n_\phi, z) \). The semantics of the future operator allows us to introduce an addend \( k \) indicating a satisfaction that happens at a current or future index: for some \( k \in \mathbb{N} \), \( \pi, i + m + k \models \text{AcycConn}(T^n_\phi, z) \). Let \( j = i + m + k \). Then, \( [y \mapsto i, z \mapsto j] \) satisfies \( L(y, z) \) and \( \pi, j \models \text{AcycConn}(T^n_\phi, z) \).

**Case 2:** \( L(y, z) \equiv [y \geq_m z] \). Rewriting the \( e_y \) function for this case, we have \( \pi, i \models X^mP \text{AcycConn}(T^n_\phi, z) \). Just as in Case 1, we move the index of satisfaction for each next (or yesterday, if \( m \leq 0 \)) operator: \( \pi, i + m \models P \text{AcycConn}(T^n_\phi, z) \). The semantics of the past operator allow us to introduce an addend \( -k \) indicating a satisfaction that happens at a current or past index: for some \( k \in \mathbb{N} \), \( \pi, i + m - k \models \text{AcycConn}(T^n_\phi, z) \). Let \( j = i + m - k \). Then, \( [y \mapsto i, z \mapsto j] \) satisfies \( L(y, z) \) and \( \pi, j \models \text{AcycConn}(T^n_\phi, z) \).

**Case 3:** \( L(y, z) \equiv [y \leq_m z, y \geq_m z] \) Rewriting the \( e_y \) function for this case, we have \( \pi, i \models \bigvee_{m \leq k \leq m'} X^k \text{AcycConn}(T^n_\phi, z) \).

Because the disjunction is true, the following holds: for some \( k \in \mathbb{Z} \), where \( m \leq k \leq m' \), \( \pi, i \models X^k \text{AcycConn}(T^n_\phi, z) \).

The semantics of the next operator allows us to move the index of satisfaction for each next (or yesterday, if \( k \leq 0 \)) operator: \( \pi, i + k \models \text{AcycConn}(T^n_\phi, z) \). Let \( j = i + k \). Then, \( [y \mapsto i, z \mapsto j] \) satisfies \( L(y, z) \) and \( \pi, j \models \text{AcycConn}(T^n_\phi, z) \).

By Lemma 4.2, these cases for \( L(y, z) \) are exhaustive, so there is some timestamp \( j \) such that each assignment that \( y \) to \( i \) and \( z \) to \( j \) satisfies \( L(y, z) \) and \( \pi, j \models \text{AcycConn}(T^n_\phi, z) \). Applying the inductive hypothesis with \( T^n_\phi \) and the timestamp \( j \), it follows that there is some assignment \( \sigma_y \) such that \( \sigma_y(z) = j \) and \( \eta \models \text{atoms}(T^n_\phi) [\sigma_y] \).

Now, we combine the assignments corresponding to each child of \( y \). Let \( \sigma \) be an assignment such that \( \sigma(y) = i \) and for each child \( z \) of \( y \), for each node \( v \) in \( T^n_\phi \), \( \sigma(v) = \sigma_z(v) \).
To finish the proof, we show $\eta \models atoms(T^x_{\phi})[\sigma]$, using three cases for the atoms $\alpha \in atoms(T^x_{\phi})$:

Case 1: $\alpha$ is a process atom using $y$. Let $\alpha = p_1@y$. By the initial assumption, $\pi_\eta, i \models \bigwedge_{p@y \in \phi} p$. Then, $\pi_\eta, i \models p_1$. Using the correspondence between enactments and traces, we have $i \in \eta(p_1)$. Using $\sigma(y) = i$, we have $\sigma(y) \in \eta(p_1)$, so $\eta \models \alpha[\sigma]$.

Case 2: $\alpha$ is a gap atom in the label $L(y, z)$ for some $z \in Ch(y)$. Because $\sigma(y) = i$ and $\sigma(z) = j$, we have that $\sigma$ satisfies $L(y, z)$, i.e. $\eta \models \alpha[\sigma]$.

Case 3: $\alpha$ is a process atom or gap atom such that all variables in $\alpha$ are in the subtree $T^x_{\phi}[z]$ for some $z \in Ch(y)$, i.e. $\alpha \in atoms(T^x_{\phi}[z])$. Applying the inductive hypothesis for $z$ yielded $\eta \models atoms(T^x_{\phi}[z])[\sigma_z]$. Then, because the assignment $\sigma_z$ is used to construct $\sigma$, $\eta \models atoms(T^x_{\phi}[z])[\sigma]$, so $\eta \models \alpha[\sigma]$.

The cases cover all atoms whose variable are nodes in $T^x_{\phi}$, so it holds that $\eta \models atoms(T^x_{\phi})[\sigma]$. Note, by the construction of $\sigma$, $\sigma(y) = i$.

A key observation in the proof of Lemma 4.7 is that an assignment $\sigma$ satisfying a constraint for a given enactment can be used to identify instants where subformulas of AcycConn become true in the enactment’s trace.

Let $\phi$ be a constraint, $x \in var(\phi)$. Note that $atoms(T^x_{\phi})[x] = atoms(T^x_{\phi}) = \phi$. Then, a corollary of Lemma 4.7 follows:

**Corollary 4.8.** Let $S$ be a service schema, $\eta$ an enactment of $S$, $\phi$ a connected, acyclic constraint over $S$, $x$ a variable in $var(\phi)$, $T^x_{\phi}$ a derived tree of $\phi$ rooted at $x$. For each timestamp $i$, the following statements are equivalent:

- For some assignment $\sigma$, $\sigma(x) = i$ and $\eta \models \phi[\sigma]$
- $\pi_\eta, i \models AcycConn(T^x_{\phi}, x)$

We now use AcycConn to define a function Acyc that translates arbitrary, possibly disconnected, acyclic constraints. Let $\phi$ be an arbitrary acyclic constraint; $\phi$ can be partitioned into $k$ connected constraints $\phi_1, \ldots, \phi_k$. Let $x_j$ be a variable in $var(\phi_j)$ for each $1 \leq j \leq k$; a translation of each $\phi_j$ is given by $AcycConn(T^x_{\phi_j}, x_j)$. For each $1 \leq j, l \leq k, j \neq q$, the sets $var(\phi_j)$ and $var(\phi_k)$ are disjoint. Thus, the assignments to these variables, and the instants where $AcycConn(T^x_{\phi_j}, x_j)$ and $AcycConn(T^x_{\phi_k}, x_k)$ are satisfied, are independent. Accordingly, we combine these translations by choosing one variable, here $x_1$, as an "anchor", translating $\phi_1$ as $AcycConn(T^x_{\phi_1}, x_1)$, then conjuncting this formula with the $AcycConn$ translations of the other connected constraints, offset by past and future operators.

The translation of $\phi$ anchored at $x_1$ is as follows:

$$Acyc(T^x_{\phi}, x_1) = AcycConn(T^x_{\phi_1}, x_1) \land \bigwedge_{2 \leq j \leq k} \left( P AcycConn(T^x_{\phi_j}, x_j) \lor F AcycConn(T^x_{\phi_j}, x_j) \right)$$

We say the translation $Acyc(T^x_{\phi}, x_1)$ is anchored at $x_1$ because when $\eta \models \phi[\sigma]$, the LTL formula $Acyc(T^x_{\phi}, x_1)$ will be true at instant $\sigma(x_1)$ in $\pi_\eta$. This anchoring is crucial in connecting the satisfaction of constraints and LTL formulas.

The following lemma extends Lemma 4.7 to acyclic constraints.

**Lemma 4.9.** Let $S$ be a service schema, $\eta$ an enactment of $S$, $\phi$ an acyclic constraint over $S$, and $x$ a variable in $\phi$. For each timestamp $i$, the following statements are equivalent:

- For some assignment $\sigma$, $\eta \models \phi[\sigma]$, $\sigma(x) = i$
- $\pi_\eta, i \models Acyc(T^x_{\phi}, x)$

Manuscript
We apply Lemma 4.7, to \( \sigma \) exists. The following example illustrates this idea.

Definition: A trio of timestamps that satisfies one gap atom. Note \( i \leq j \)

Recall that an enactment \( \eta \) satisfies a rule \( \phi \rightarrow \psi \) if for every assignment \( \sigma \) where \( \eta \models \phi[\sigma] \), there is some assignment \( \sigma' \) that extends \( \sigma \) such that \( \eta \models \psi[\sigma'] \). Because \( \phi \rightarrow \psi \) is simple, \( \phi \) and \( \psi \) share (at most) one variable. The key idea in translating rules is joining the LTL translations of \( \phi \) and \( \psi \) at an instant corresponding to their common variable (if it exists). The following example illustrates this idea.

Example 4.11. Consider the constraint \( \phi_{\text{RENTAL}} \) introduced in Example 4.1 and a new constraint \( \phi_{\text{BILLING}} = \{ \text{Payment}@u, \text{Receipt}@v, u \leq_0 v, y \geq_2 u \} \), satisfied when a Payment process instance is simultaneous with or followed by a Receipt, and the Payment

**Proof.** We prove Lemma 4.9 by with logical implication. Let \( i \) be an arbitrary timestamp. Let \( \phi_1, \ldots, \phi_k \) be connected constraints such that \( \phi_1, \ldots, \phi_k \) is a partition of \( \phi \). Let \( x_j \) be a variable in \( \phi_j \) for \( 1 \leq j \leq k \). Without loss of generality, assume \( x = x_1 \).

Assume that for some assignment \( \sigma, \eta \models (\phi_1 \cup \ldots \cup \phi_k)[\sigma] \) and \( \sigma(x_1) = i \). Then, \( \eta \models \phi_1[\sigma] \) and \( \sigma(x_1) = i \).

We apply Lemma 4.7, to \( \phi_1 \) with the derived tree \( T_{\phi_1}^x \) and instant \( i \), yielding \( \pi_\eta, i_1 \models \text{AcycConn}(T_{\phi_1}^x, x_1) \). For each \( 2 \leq j \leq k, \eta \models \phi_j[\sigma] \) and for some \( i_j \in \mathbb{N}, \sigma(x_j) = i_j \). We apply Lemma 4.7 to \( \phi_j \) with derived tree \( T_{\phi_j}^y \), and instant \( i_j \), yielding \( \pi_\eta, i_j \models \text{AcycConn}(T_{\phi_j}^y, x_j) \). Because \( \text{var}(\phi_1) \) and \( \text{var}(\phi_j) \) are disjoint for \( j \neq i \), the instants \( i_1 \) and \( i_j \) are independent. Thus, the values \( i_2, \ldots, i_k \) are arbitrarily ordered with respect to \( i_1 \) (and each other), so for each \( 2 \leq j \leq k \), we have \( \pi_\eta, i_1 \models \phi(P \text{AcycConn}(T_{\phi_j}^y, x_j) \lor F \text{AcycConn}(T_{\phi_j}^y, x_j)) \). Combining these statements with a conjunction and adding \( \pi_\eta, i_1 \models \phi(P \text{AcycConn}(T_{\phi_j}^y, x_1)) \) yields \( \pi_\eta, i \models \phi(T_{\phi_j}^y, x) \).

Proving the converse uses identical reasoning.

**Example 4.10.** Consider the constraint \( \psi_{\text{PAID}} = \{ \text{Request}@x, \text{Schedule}@y, \text{Payment}@z, x \leq_3 y \} \) over \( S_{\text{RM}} \) that selects a trio of timestamps that satisfy one gap atom. Note \( \psi_{\text{PAID}} \) is acyclic but not connected, and can be partitioned into connected constraints: \( \psi_1 = \{ \text{Request}@x, \text{Schedule}@y, x \leq_3 y \} \) and \( \psi_2 = \{ \text{Payment}@z \} \). Applying \( \text{AcycConn} \) produces

\[
\text{AcycConn}(T_{\psi_1}^x, x) = \text{Request} \land X^3 \text{F Schedule}, \quad \text{and} \quad \\
\text{AcycConn}(T_{\psi_2}^z, x) = \text{Payment}
\]

Picking \( \psi_1 \) and \( x \) to the anchor the translation yields

\[
\text{Acyc}(T_{\psi_1}^x, x) = \text{AcycConn}(T_{\psi_1}^x, x) \land \left( (P \text{AcycConn}(T_{\psi_2}^z, z) \lor (F \text{AcycConn}(T_{\psi_2}^z, z)) \right)
\]

\[
= (\text{Request} \land X^3 \text{F Schedule}) \land \left( (P \text{Payment}) \lor (F \text{Payment}) \right)
\]

Now we turn to the main technical result of our paper: translations of a subclass of rules.

**Definition:** A rule \( \phi \rightarrow \psi \) over a service schema is simple if \( \phi \cup \psi \) is acyclic and at most one variable is shared by \( \text{var}(\phi) \) and \( \text{var}(\psi) \).

Recall that an enactment \( \eta \) satisfies a rule \( \phi \rightarrow \psi \) if for every assignment \( \sigma \) where \( \eta \models \phi[\sigma] \), there is some assignment \( \sigma' \) that extends \( \sigma \) such that \( \eta \models \psi[\sigma'] \). Because \( \phi \rightarrow \psi \) is simple, \( \phi \) and \( \psi \) share (at most) one variable. The key idea in translating rules is joining the LTL translations of \( \phi \) and \( \psi \) at an instant corresponding to their common variable (if it exists). The following example illustrates this idea.

**Example 4.11.** Consider the constraint \( \phi_{\text{RENTAL}} \) introduced in Example 4.1 and a new constraint \( \phi_{\text{BILLING}} = \{ \text{Payment}@u, \text{Receipt}@v, u \leq_0 v, y \geq_2 u \} \), satisfied when a Payment process instance is simultaneous with or followed by a Receipt, and the Payment
The rule TimelyPayment is expressed as: \( \varphi_{\text{rental}} \rightarrow \varphi_{\text{billing}} \). This rule is simple because the union of its constraints is acyclic and \( y \) is the only variable the constraints share. Note that \( \varphi_{\text{rental}} \) is closed but \( \varphi_{\text{billing}} \) is not. However, \( \varphi'_{\text{billing}} = \varphi_{\text{billing}} \cup \{ \text{Schedule}@y \} \) is closed. In Fig. 5, the graphs of \( \varphi_{\text{rental}} \) and of \( \varphi'_{\text{billing}} \) are shown, with a dotted line indicating that \( \varphi'_{\text{billing}} \) has been extended with the process atom from the shared variable \( y \).

We translate \( \varphi_{\text{rental}} \) and \( \varphi'_{\text{billing}} \) using the \( \text{Acyc} \) function anchored at \( y \):

\[
\text{Acyc}(T^y_{\text{rental}}, y) = \text{Schedule} \land \left( \bigvee_{-10 \leq j \leq -1} X^j \right) \land X^7 P \left( \text{Compute} \land \text{F Terminate} \right), \quad \text{and}
\]

\[
\text{Acyc}(T^y_{\text{billing}}, y) = \text{Schedule} \land X^7 P \left( \text{Payment} \land \text{F Receipt} \right)
\]

TimelyPayment expresses a property of all assignments that satisfy \( \varphi_{\text{rental}} \). Using Lemma 4.9, this corresponds to a property of all instances of a trace that satisfy \( \text{Acyc}(T^y_{\text{rental}}, y) \). The property is an expectation of all instances of a trace that satisfy \( \text{Acyc}(T^y_{\text{rental}}, y) \). We use the implication operator to reflect this expectation. To reflect the possibility of \( y \) being assigned an arbitrary timestamp, we place the implication in the scope of the global operator. The translation of TimelyPayment is

\[
\ell^{\text{TL}}_{\text{TimelyPayment}} = G(\text{Acyc}(T^y_{\text{rental}}, y) \rightarrow \text{Acyc}(T^y_{\text{billing}}, y))
\]

The translation shown in Example 4.11 generalizes to a translation function \( \gamma \), below, for simple rules. Let \( \phi \rightarrow \psi \) be a simple rule. In the first case, assume \( \phi \) and \( \psi \) share a variable \( x \). Let \( \psi' \) be the union of \( \psi \) and the set of process atoms in \( \phi \) with \( x \). Note that \( \psi' \) is closed. We obtain translations \( \text{Acyc}(T^x_{\phi'}, x) \) and \( \text{Acyc}(T^x_{\psi'}, x) \) of \( \phi \) and \( \psi' \) (resp.) anchored at \( x \). Then the translation \( \gamma(\phi \rightarrow \psi) \) is \( G(\text{Acyc}(T^x_{\phi'}, x) \rightarrow \text{Acyc}(T^x_{\psi'}, x)) \). In the second case, when \( \phi \) and \( \psi \) have no common variables, the translation is \( \gamma(r) = G(\text{Acyc}(T^x_{\phi'}, x) \rightarrow \left( \text{P Acyc}(T^z_{\phi'}, z) \lor \text{F Acyc}(T^z_{\phi'}, z) \right)) \), where \( x \) is a variable in \( \phi \), \( z \) a variable in \( \psi \).

\[
\gamma(\phi \rightarrow \psi) = \begin{cases} 
G(\text{Acyc}(T^x_{\phi'}, x) \rightarrow \text{Acyc}(T^x_{\psi'}, x)) & \text{if } \phi \text{ and } \psi \text{ share some } x \\
G(\text{Acyc}(T^x_{\phi'}, x) \rightarrow \left( \text{P Acyc}(T^z_{\phi'}, z) \lor \text{F Acyc}(T^z_{\phi'}, z) \right)) & \text{otherwise, where } z \in \text{var}(\psi) 
\end{cases}
\]

Fig. 6. Translation \( \gamma \) of simple rule \( \phi \rightarrow \psi \)

The following theorem establishes the correctness of the translation function \( \gamma \).

**Theorem 4.12.** Let \( S \) be a service schema, \( \eta \) an enactment of \( S \), \( \phi \rightarrow \psi \) a simple rule. Then, the following statements are equivalent:

- \( \eta \models \phi \rightarrow \psi \)
- \( \pi_\eta, 1 \models \gamma(\phi \rightarrow \psi) \)

and it follows directly that for a set of rules \( R \), the following statements are equivalent:

- \( \eta \models R \)
- \( \pi_\eta, 1 \models \bigwedge_{r \in R} \gamma(r) \)

**Proof.** We prove the first equivalence of Theorem 4.12 by showing both directions of logical implication.

Assume \( \eta \models \phi \rightarrow \psi \). We consider two cases for the variables in \( \phi, \psi \).
Case 1: $\phi$ and $\psi$ share some variable $x$. In this case, $\phi \rightarrow \psi$ is translated as $\gamma(\phi \rightarrow \psi) = G(\text{Acyc}(T^x_{\phi}, x) \rightarrow \text{Acyc}(T^x_{\psi}, x))$. To prove $\pi_\eta, 1 \models G(\text{Acyc}(T^x_{\phi}, x) \rightarrow \text{Acyc}(T^x_{\psi}, x))$, it is sufficient to show that for every instant $i$ of the trace, if $\pi_\eta, i \models \text{Acyc}(T^x_{\phi}, x)$, then $\pi_\eta, i \models \text{Acyc}(T^x_{\psi}, x)$. Let $i$ be an arbitrary instant of $\pi_\eta$. Assume $\pi_\eta, i \models \text{Acyc}(T^x_{\phi}, x)$. Applying Lemma 4.9, there is an assignment $\sigma$ such $\eta \models \phi[\sigma]$ and $\sigma(x) = i$. Because, by assumption, $\eta$ satisfies $\phi \rightarrow \psi$, there is some assignment $\sigma'$ that extends $\sigma$ such that $\eta \models \psi[\sigma']$, i.e. $\sigma'(x) = i$. We apply Lemma 4.9 with the satisfaction $\eta \models \psi[\sigma']$ and $\sigma'(x) = i$, yielding $\pi_\eta, i \models \text{Acyc}(\psi, x)$, concluding this case.

Case 2: $\text{var}(\phi) \cap \text{var}(\psi) = \{x\}$, where $x \in \text{var}(\phi)$, $z \in \text{var}(\psi)$. In this case, $\phi \rightarrow \psi$ is translated as $\gamma(\phi \rightarrow \psi) = G(\text{Acyc}(T^x_{\phi}, x) \rightarrow (P \text{Acyc}(T^z_{\phi}, z) \lor \text{Facyc}(T^z_{\psi}, z)))$. To show $\pi_\eta, 1 \models G(\text{Acyc}(T^x_{\phi}, x) \rightarrow (P \text{Acyc}(T^z_{\phi}, z) \lor \text{Facyc}(T^z_{\psi}, z)))$, it is sufficient to show that if $\text{Acyc}(T^x_{\phi}, x)$ is satisfied at some instant $i$ of the trace, then $(\text{Facyc}(T^z_{\phi}, z)) \lor \text{Facyc}(T^z_{\psi}, z)$ is satisfied at $i$ as well, i.e. that $\text{Acyc}(T^z_{\phi}, z)$ is satisfied somewhere in the trace. Let $i$ be an arbitrary instant of $\pi_\eta$. Assume $\pi_\eta, i \models \text{Acyc}(T^x_{\phi}, x)$. By Lemma 4.9, there is an assignment $\sigma$ such that $\eta \models \phi[\sigma]$ and $\sigma(x) = i$. Because it is assumed that $\eta \models \phi \rightarrow \psi$, there is an assignment $\sigma'$ such that $\eta \models \psi[\sigma']$ and $\sigma'(z) = j$. Applying Lemma 4.9 to transform this satisfaction, we have $\pi_\eta, j \models \text{Acyc}(\psi, z)$. Either $i \geq j$ and thus $\pi_\eta, i \models P\text{Acyc}(\psi, z)$ or $i \leq j$ and thus $\pi_\eta, i \models \text{Facyc}(T^z_{\psi}, z)$. In either case, the sufficient condition is established.

By definition, all simple rules have constraints that share no variables (Case 1) or one variable (Case 2), so these cases are exhaustive. Thus, the statement that $\eta \models \phi \rightarrow \psi$ logically implies $\pi_\eta, 1 \models \gamma(\phi \rightarrow \psi)$ has been shown.

The converse, stating that $\eta \models \phi \rightarrow \psi$ follows from $\pi_\eta, 1 \models \gamma(\phi \rightarrow \psi)$, is similar.

The second equivalence a set of rules is interpreted as a conjunction, matching the semantics of the LTL conjunction operator.

The proof of Theorem 4.12 relies on the construction in the proof of Lemma 4.9 that connects a satisfying assignment for a constraint with one instant in the trace where the constraint’s translation is satisfied. The variable $y$ is shared between $\phi$ and $\psi$, so the translations $\text{Acyc}(T^y_{\phi}, y)$ and $\text{Acyc}(T^y_{\psi}, y)$ are each anchored at $y$. This anchoring ensures that satisfying assignments to $\phi$ are matched with satisfying assignments to $\psi$ such that they assign the same timestamp to $y$.

5 RELATED WORK

This paper is closely related to the work on monitoring BPEL services with rules specified in event calculus [24, 33]. In particular, [33] employed an event calculus reasoning engine to recognize violations of rules expressed in event calculus. In both approaches, the size of a monitor (an event database) is unbounded, while in our approach, a monitor has a fixed size that depends only on the size of the rule. [2] used past- and future-time LTL and developed an interesting formula progression technique for monitoring. [27] proposed monitoring rules expressed in EaGLe for a service composition using finite state machines, with no support for quantitative time constraints.

Our work is also related to monitoring runtime behaviors of web services and business processes specified in temporal logics. For example, [23] turned Declare constraints [26] into colored finite state automata for runtime verification. [9] mapped LTL and Linear Dynamic Logic into finite state machines to monitor business process executions. Papers [10, 16] mapped first-order future-time LTL to Büchi automata for runtime monitoring. Note that (first-order) LTL does not naturally support quantitative time constraints. Unlike these logics, our language allows for specification of quantitative and past-time rules. [25] extended the Declare framework with quantitative time constraints and map
this extension into event calculus to monitor constraints on business processes. For general software systems, [17] generated binary decision diagrams for monitoring first-order past LTL properties.

A less related body of work is the study on controllability of web services, e.g., [7, 12, 18]. Although explicit time instants are used but the controllability problem is quite different from runtime monitoring.

6 CONCLUSIONS

This paper initiates a study on service provisioning with an emphasis on modeling relationships between business processes in a service. A solution based on runtime monitoring techniques with finite state machines derived from temporal logic is developed. There are several problems regarding service provisioning that deserve more research. First, service modeling in particular development of specification languages is interesting. Understanding issues such as explicit timestamps (e.g. rules in this paper) vs implicit timestamps (e.g., LTL), past constraints vs future constraints, will lead to more suitable specifications for service applications. Also, runtime monitoring techniques are far from satisfactory. Translating classes of rules richer than simple rules, e.g. rules with data deserves further study. Also, it may be possible to directly translate rules to automata of smaller size. Formula progression techniques developed in [2] seem to be an interesting alternative.

REFERENCES