**myjournal manuscript No.**
(will be inserted by the editor)

# A Logical Approach to Data-Aware Automated Sequence Generation

**Sylvain Hallé · Roger Villemaire · Omar Cherkaoui · Rudy Deca**

Received: 7 July 2010 / Accepted: 29 October 2010

**Abstract** Automated sequence generation can be loosely defined as the algorithmic construction of a sequence of objects satisfying a set of constraints formulated declaratively. A variety of scenarios, ranging from self-configuration of network devices to automated testing of web services, can be described as automated sequence generation problems. In all these scenarios, the sequence of valid objects and their data contents are interdependent. Despite these similarities, most existing solutions for these scenarios consist of *ad hoc*, domain-specific tools. This paper stems from the observation that, when such "data-aware" constraints are expressed using mathematical logic, automated sequence generation becomes a case of satisfiability solving. This approach presents the advantage that, for many logical languages, existing satisfiability solvers can be used off-the-shelf. Te paper surveys three logics suitable to express real-world data-aware constraints and discusses the practical implications, with respect to automated sequence generation, of some of their theoretical properties.

## 1 Introduction

Historically, the advancement of computing has been marked by the development of successive abstractions in the description of the tasks to be accomplished by a system. As an example, in the field of programming, the advent of the assembly language, followed by structured programming languages, allowed users to progressively distance themselves from technical and hardware issues to concentrate on a logical description of the work to be done.

S. Hallé
Université du Québec à Chicoutimi, Canada
E-mail: shalle@acm.org

R. Villemaire, O. Cherkaoui, R. Deca
Université du Québec à Montréal, Canada
E-mail: {villemaire.roger, cherkaoui.omar}@uqam.ca

This evolution towards more abstract descriptions continues to this day. Until recently, the development and use of a system followed an approach which could be called *imperative*: although the languages and the formalisms evolved towards more abstract concepts, the basic principle always consisted of describing the tasks to be carried out in order to produce a desired result. However, a number of fields in computing have been transformed in recent years with the advent of a new, *declarative* approach.

In a declarative paradigm, the results, rather than the tasks, are described by means of a language. The actual way in which these results should be obtained is not specified. The new job for a user is no longer to design and express a sequence of steps that the system should follow, but rather to describe as precisely as possible the data or functionalities expected from the system. While the imperative approach "commands", the declarative approach "demands".

This raises a fundamental problem: to develop automated techniques to produce a result that satisfies some declarative specification. The premise of this work is the following: mathematical logic represents an appropriate formal foundation for the resolution of this problem. Indeed, mathematical logic is essentially a declarative language: by means of logical connectives, it is possible to build statements that become true or false depending on the context on which they are interpreted.

First, in Section 2, the paper studies the concept of sequential constraints in computing and shows by a couple of examples their widespread presence in a variety of scenarios, ranging from web service compositions to the management of network devices. Section 2.3 then shows how the automated generation of sequences of operations can be put to numerous uses in the aforementioned scenarios. Depending on the field, automated sequence generation becomes an instance of self-configuration, test case generation, or web service composition problems.

Second, in Section 3, the paper provides a simple formal model for representing operations and sequences of operations carrying simple data payloads such as command parameters, XML message elements, or function arguments. It shows how all the presented scenarios can be appropriately represented under this model, thereby giving a common, domain-independent formal foundation for the study of sequential dependencies in computing.

Finally, the paper advocates the use of logical methods for representing such constraints in a declarative fashion. Automated sequence generation then becomes a mere instance of satisfiability solving for a set of logic formulæ. An interesting consequence is that, for many logics, automated satisfiability solvers are readily available, and can be used as general purpose engines, thus removing the need for *ad hoc*, domain-specific solutions. In addition, well studied properties of these logics, such as undecidability theorems or complexity results, apply directly to the computing scenarios they are used to model.

This formal model reveals an important and often overlooked characteristic of sequential properties. In many cases, the sequence of valid objects and their respective data contents cannot be treated separately. Yet, we shall see that traditional logics are generally ill-equipped for expressing such kinds of constraints.

To the best of the authors' knowledge, this is the first attempt at a domain-independent study of logic-based automated sequence generation. Hence, the paper intends to be a road map for future work on that topic. To this end, a set of three logics suitable for that task are described in Sections 4 to 6. The same sample properties of a real-world scenario are modelled in each of them; tools and techniques for automatically generating sequences satisfying the resulting formulæ are then presented.

## 2 Sequential Aspects in Computing

The first part of this paper deals with the description of valid sequences of operations in various fields, and describes how such sequences of operations can be automatically generated. We first illustrate the concept of sequential dependencies by means of two examples taken from networks and web services. For each of these examples, sequential dependencies are extracted and formalized.

### 2.1 Example 1: Management of Network Services

Computer networks have become over the years the central element of numerous applications. While they were originally limited to a few basic functionalities (e-mail, file transfer), a plethora of new services are adding to the value of these networks: voice over IP, virtual private networks (VPNs), peer-to-peer communities. The growth of these functionalities over the years considerably increased the complexity of managing the devices responsible for their proper functioning, in particular network routers [62].

The deployment of a service over a network basically consists in altering the configuration of one or many equipments to implement the desired functionalities. We take as an example the Virtual Private Network (VPN) service, which consists in a private network constructed within a public network such as a service provider's network [60].

Such a service consists of multiple configuration operations; in the case of so-called "Layer 3 VPNs", it involves setting the routing tables and the VPN forwarding tables, setting the MPLS, BGP and IGP connectivity on multiple equipments having various roles. An average of 10 parameters must be added or changed in each device involved in the deployment of a VPN.

Figure 1 shows a sequence of commands in the Cisco Internet Operating System (IOS) for the configuration of a Virtual Routing and Forwarding table (VRF) on a router. Command `ip vrf` is to be executed first. Its purpose is to create a VRF instance for the *Customer_1* VPN on a router. This command also opens the `ip vrf` configuration mode, in which the main VRF parameters can be configured. These parameters are the *route distinguisher* (`rd`), which

```
ip vrf Customer_1
rd 100:110
route-target both 100:1000
ip vrf forwarding Customer_1
```

**Fig. 1** A sequence of VRF configuration commands for the VPN example.

allows the unique identification of the VPN's routes throughout the network, and the `import/export route target`, which are exchanged between ingress and egress routers in order to identify the VPNs to which the updates belong to [56, 60].

After the creation of the VRF, it is activated on one of the router's interfaces. However, when done in an uncoordinated way, changing, adding or removing components or data that implement network services can bring the network in an inconsistent or undefined state.

In this context, the application of a declarative approach becomes highly desirable. For a network manager, it consists of describing the services or the functionalities available on a network, in conjunction with the modalities for consuming these services or functionalities by the users (customers). To this end, the official documentation for VRF functions in Cisco's operating system [4] describes a clear sequential order of execution among these VRF configuration commands. Firstly, we cannot configure the `rd` and `route-target` parameters of a VRF before creating it with the command `ip vrf`:

**Sequential Rule 1** *Command* `rd` *must be called after command* `ip vrf`*.*

Secondly, we cannot activate a VRF on an interface before creating it (with command `ip vrf`) and configuring its parameters (with commands `rd` and `route target`). Moreover, the VRF we activate must be the one we created, so there is also a relationship between the VRF name in both commands:

**Sequential Rule 2** *Command* `ip vrf forwarding` *must be called after commands* `rd` *and* `route-target`*. Moreover, it must have the same argument as* `ip vrf`*.*

Note that other sequential rules could be similarly expressed for the `route-target` commands and between the `ip vrf forwarding`. The reader is referred to [56] for further details about the setup of a VPN in Cisco's IOS.

### 2.2 Example 2: Transactional Web Services

As a second example, we take the case of increasingly popular web applications, where some server's functionality is made publicly available as an instance of a *web service* that can be freely accessed by any third-party application running in a web browser. Notable examples of applications using such a scheme include Facebook, Twitter, and Google Maps, among others. In most cases, the communication between the application running in the browser and the web

```
<message>
  <action>getStockDetails</action>
  <stocks>
    <stock-name>123</stock-name>
    <stock-name>456</stock-name>
    <stock-name>789</stock-name>
  </stocks>
</message>
```

**Fig. 2** A sample XML message sent by a web application to a web service.

service is done through the exchange of documents formatted in the eXtensible Markup Language (XML), as shown in Figure 2.

The precise format in which web service operations can be invoked by an applications is detailed in a a document called an *interface specification*; for XML-based interactions, the Web Service Description Language (WSDL) [14] provides a way of defining the structure and acceptable values for XML requests and responses exchanged with a given service.

While such "data contracts" are relatively straightforward to specify and verify, interface contracts go beyond such static requirements and often include a temporal aspect. For example, the online documentation for the popular Amazon E-Commerce Service [2] elicits constraints on possible *sequences* of operations that must be fulfilled to avoid error messages and transaction failures [40].

As a simple example, we consider the case of an online trading company, taken from [34, 42]. The company is responsible for selling and buying stocks for its customers; it exposes its functionalities through a web service interface. An external buyer (which can be a human interfacing through a web portal, or another web service acting on behalf of some customer) can first connect to the trading company and ask for the list of available products. This is done through the exchange of a getAllStocks message, to which the company replies with a stockList message. The customer can then decide to get more information about each stock, such as its price and available quantity, using a getStockDetails message giving the list of stock names on which information is asked (see Figure 2). The trading company replies with a stockDetails message listing the information for each stock. Finally, the customer can buy or sell stock products. In the case of a buy, this is done by first placing a placeBuyOrder message, listing the name and desired amount of each products to be bought.

This scenario also comprises constraints on the possible sequence of valid operations. For example, a user can call a cash transfer by sending a cashTransfer message without having first specified whether a stock should be bought or sold, or which precise stock is concerned by the transaction. A first choreography constraint would then be:

**Sequential Rule 3** *No cash transfer can be initiated before a buy or sell confirmation has been issued.*

A guarantee on the termination of pending transactions can also be imposed. More precisely, the system can require that all transactions eventually complete in two possible ways:

**Sequential Rule 4** *Every buy/sell order is eventually completed by a cash transfer or a cancellation referring to the same bill ID.*

Although there exist other constraints in this scenario, these two sequential rules are sufficient for the paper's thesis. For a more thorough study of sequential constraints in transactional web services (and additional examples), the reader is referred to [32, 35].

## 2.3 Applications of Automated Sequence Generation

The previous scenarios share two important points. First, they involve sets of discrete "events": in the first scenario, events are configuration commands; in the second scenario, events are web service messages sent or received. Second, these events occur in a precise (or at least, not completely arbitrary) *sequence*. Therefore, as we have seen, configuring a VPN not only involves figuring out what commands to issue to a router; these commands will be rejected if they are issued in the "wrong" order. Similarly, interacting with the online trading company will only succeed if the protocol imposed by the web service is respected by both sides.

The prospect of automatically generating sequences of such events, given a set of declarative constraints similar to the previous Sequential Rules, is appealing. As it turns, this concept finds many applications in various fields. We briefly mention a few of them, along with pointers to additional resources for each.

### *2.3.1 Self-Configuration in Autonomic Computing*

The automated generation of a structure satisfying a set of properties is part of a more general paradigm called *autonomic computing* [37, 55]; applied in particular to computer networks, this notion is called *autonomic* (or autonomous) *networking* [25].

Self-configuration and self-healing are two central capabilities that a network element must implement in order to exhibit autonomic behaviour [55]. In *self-configuration*, the numerous parameters of a device are populated with appropriate values without immediate external intervention; these values depend both on the element's role and its network context. Narain [52] describes a typical self-configuration architecture, called a *service grammar*. In this context, the requirements language expresses constraints relative to each configuration command; a "synthesis engine" produces commands and appropriate values that fulfil these requirements, which are then applied to each device. The VPN scenario described in Section 2.1 corresponds to this setting: an automatically generated sequence of commands "auto-configures" a VPN.

Most existing configuration management systems, such as Cfengine [13] and Bcfg2 [21], use an imperative rather than declarative approach, and are therefore only partially appropriate for the present problem. An approach suggested by Narain [53] uses a Boolean satisfiability solver to produce a configuration for a set of routers satisfying a number of network constraints expressed in the Alloy language [41]. Such an approach was also explored for the automated generation of configuration parameters in network devices, using different logics, by the authors in [36].

However, none of these solutions involve sequential constraints, and only refer to static snapshots of configurations. As the previous scenarios showed, in many cases finding the appropriate operations and values for these operations is not sufficient.

### *2.3.2 Automated Composition of Web Services*

An open challenge in the web service field is to automatically find a way to make multiple web services interact between each other without a predefined pattern of interaction: this is called *web service choreography* [6]. In such a situation, a script involving all the potential web services is generated dynamically from a set of specifications for each service. It is up to the composition engine to resolve these constraints and call each services' operations in ways that are consistent with each set of requirements.

The main motivation for this setting is the fact that web services forming a choreography can be selected dynamically, at runtime. The classical example is a travel agency, which asks for quotes from various providers before dynamically selecting the best offer and combining reservations details for a car, a hotel room and airline tickets. Clearly, it is impractical to compute in advance all possible combinations of web service providers, and hence their interaction should be guided by the sum of their individual "protocols".

The scenario shown in Section 2.2 corresponds to this setting. The online trading company publicizes a number of data and sequential constraints on its possible invocations. Any potential client must abide by these constraints in order to successfully interact with it.

The automated composition of web services is a very popular research field that has spawned a large number of works over the past decade. Notable contributions include the use of semantic web technologies [54], and in particular *ontologies*, to provide consistent theories of a particular domain intended as a common language for all potential partners. These ontologies can be used, among other things, to describe pre- and post-conditions on operations similar to sequential constraints. Many ontology languages, such as the Ontology Web Language (OWL) [48], use logic as the underlying foundation for expressing their base concepts; automated tasks can then be accomplished on web services annotated with OWL documents using reasoners.

However, the problem can hardly be considered solved, and a consensus has yet to be reached on the way each service's requirement should be expressed, and up to which level of detail. In addition, many ontologies use a variant of first-order logic as their formal language; as we shall see, some properties of first-order logic (namely undecidability) warrant its careful use. Good surveys on existing techniques in web service composition can be found in [18, 58].

### *2.3.3 Automated Stub Generation*

Often a web service under development cannot be run and tested in isolation, and must communicate with its actual partners, even in its early stages of design. Yet for various reasons, it might be desirable that no actual communication takes place. For example, one might want to control the possible responses from the outside environment to debug some functionality; or one might prefer not to provoke real operations (such as buying or selling products) on the actual third-party services during the testing phase.

To this end, a web service "stub" can be constructed. This stub is a kind of mock-up web service, acting as a placeholder for an actual one and simulating its input-output patterns. The degree of "faithfulness" of these mock-ups can vary: sometimes it suffices to return the same response for all inputs of a certain type, while at other times a finer granularity is desirable; the example in Section 2.2 shows that a realistic stub for the online trading company should also simulate transactions involving sequential constraints over multiple operations.

This particular use case of automated sequence generation brings the concept of *interactivity*. Since the stub is composed of inputs sent by a third-party service, and returns responses to these inputs, the sequence of operations is constructed in an incremental fashion. Therefore, at any point in time, an automated generator must not only produce a sequence that satisfies the numerous sequential rules, but must exhibit a trace which is a valid *extension* of a given prefix.

Currently, web service stubs need to be coded by hand, and are hence specific to each development project. Yet, as was pointed in [40], automatically generating such stubs based on a declarative specification of the service would relieve programmers of a task they often overlook. A commercial development tool for web services, called soapUI [3], allows a user to create "mock web services", which consist of hard-coded responses for specific inputs. A similar tool called WebSob [49] can generate random requests and discover incorrect handling of nonsensical data on the service side. However, all these approaches treat request-responses as atomic patterns independent of each other. Of all works, only [32] considers an automated generation of sequences of messages, using a variant of formal grammars.

### *2.3.4 Test Case Generation*

Application testing in a regular operating system can be made by simulating a user generating GUI events. A well-known testing technique, called *monkey testing* [8], involves a testing script generating random sequences of GUI events. However, these events have to be consistent with the state of the application, and hence constraints on the sequence of possible events have to be followed. For example, it does not make sense to send a user interface event for a window that has just been closed: a system crash resulting from that event would not indicate a flaw in the application under test, since the application itself would not allow such a sequence of events to be produced in the first place.

Therefore, an educated monkey tester should generate random sequences of events, taken from the set of all actions that can be produced; as we see, such actions depend on their ordering. An automated sequence generator, provided with a formalization of typical GUI behaviours, provides such a tester. The field of *automated test case generation* concentrates on the production of meaningful test cases for applications under development; in particular, model-based testing attempts to generate these cases from formal models of the system under test. Some approaches use Boolean satisfiability solvers to this end, in the manner of the present paper [44].

## 3 Automated Sequence Generation Through Logic

The previous section has shown how a single concept, namely the automated generation of sequences of operations according to some declarative specification, corresponds to a wide variety of tasks depending on its field of application. In all these fields, the corresponding problem is still considered open. The presentation of related work hence highlights the presence of multiple, seemingly *ad hoc* and domain specific solutions tailored for particular subproblems.

### 3.1 A Formal Model of Automated Sequence Generation

Yet, the previous scenarios share important common points. They can be modelled as sequences of "events", which contains not only a name, but also a series of parameters and associated values. For example, configuration commands consist of a command name, and of one or more arguments. In the same way, web service messages have an XML structure that can be likened to a command and a list of parameters. From this observation, it is possible to deduce a common formal model, represented in Figure 3.

The interaction with an object (upper box) is carried by a sequence of operations sent or received (represented by the documents at the bottom). Each of these operations is a tuple $(a, \star)$, where $a$ is a label (represented by letters A-D), and $\star$ is a "content". Formally, let $P$ be a set of parameters, $V$ be a set of values, and $A$ be a set of action names. We define the set of operations as $O \subseteq A \times 2^{P \times 2^V}$ as a set of tuples $(a, \star)$, where $a \in A$ is an action and $\star$ is a relation associating to each parameter $p \in P$ a subset of $V$.

In the context of network configurations, the operation label can represent the name of a command or of a block of commands to execute; the set of parameter-value pairs represents the parameters of the operation, and in particular the configuration or the portion of configuration the operation acts

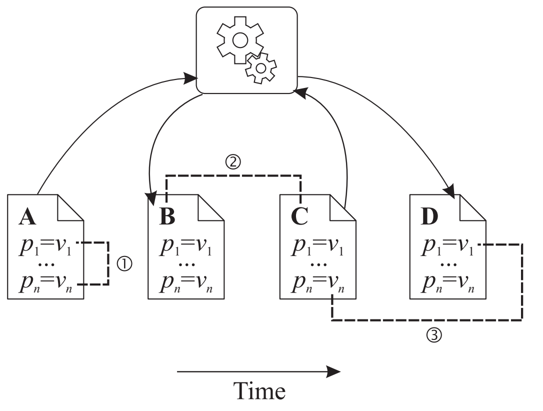

**Fig. 3** A formal model to represent declarative constraints in computing. Dashed lines numbered 1, 2, 3 respectively represent static, temporal, and "data-aware" constraints over traces of operations.

upon. Previous works have shown that such structures are an appropriate abstraction of configuration information and operations in network devices [31]. For example, in the VPN scenario, we can define these sets as:

$$A = \{\texttt{ip vrf}, \texttt{rd}, \texttt{route-target}, \texttt{ip vrf forwarding}\}$$
$$P = \{\texttt{vrf-name}, \texttt{as}, \texttt{send-community}\}$$
$$V = \{\texttt{100:110}, \texttt{200:210}, \texttt{both}, \texttt{import}, \texttt{export}, \texttt{Customer\_1}, \texttt{Customer\_2}\}$$

Using this representation, configuration commands can be represented as operations. For example, the IOS command `neighbor 10.1.2.3 send-community both` can be represented as an operation $o = (\texttt{neighbor}, \star)$, where $\star$ is the relation such that $\star(\texttt{address}) = \{10.1.2.3\}$ and $\star(\texttt{send-community}) = \{\texttt{both}\}$.

Obviously, some parameters are only appropriate for some operations, and some values are only appropriate for some parameters. For example, the `ip vrf` operation only takes a VRF name, and not a `send-community` parameter. Similarly, it does not make sense for the VRF name to have as a value an IP address such as 10.1.2.3. Therefore, in addition to the previous sets, one can define two ancillary functions:

– A *schema* function $S : A \to 2^P$ indicating which parameters can appear in an operation for a given action
– A *domain* function $D : P \to 2^V$ indicating the possible values, among the set $V$, are allowed for each parameter.

In the VPN example, one can define functions $S$ and $D$ as shown in Table 1.

Similarly, in the context of service oriented architecture, the label represents the name of an XML message or of an operation, and the parameter-value pairs represents the content of the XML message. We omit the formalization

$$\begin{aligned} S = \{\ & (\texttt{ip vrf}, \{\texttt{vrf-name}\}),\\ & (\texttt{rd}, \{\texttt{as}\}),\\ & (\texttt{route-target}, \{\texttt{route-distinguisher}, \texttt{as}\}),\\ & (\texttt{ip vrf forwarding}, \{\texttt{vrf-name}\})\} \end{aligned}$$

$$\begin{aligned} D = \{\ & (\texttt{vrf-name}, \{\texttt{Customer\_1}, \texttt{Customer\_2}\}),\\ & (\texttt{rd}, \{\texttt{100:110}, \texttt{200:210}\}),\\ & (\texttt{send-community}, \{\texttt{both}, \texttt{import}, \texttt{export}\})\} \end{aligned}$$

**Table 1** Schemas and domains definitions for the VPN example.

of the web service scenario, which can be translated in a similar manner as VPN commands.

## 3.2 Automated Sequence Generation Through Satisfiability Solving

A *trace* of operations is a sequence $\bar{o} = o_0, o_1, \ldots$, where $o_i \in O$ for each $i \geq 0$. *Automated sequence generation* can be loosely defined as the algorithmic generation of a trace of operations satisfying a set of constraints expressed declaratively. In the present context, an automated sequence generation architecture can be mapped to some formal concepts. Given:

– some language $\mathcal{L}$ to describe constraints
– a sequence of operations $\bar{o}$ whose values need to be found
– $\varphi$, a description of the constraints on $\bar{o}$ expressed in the language $\mathcal{L}$
– $P$, a procedure that finds values for $\bar{o}$ that comply with $\varphi$

then the computation of $P(\varphi, \bar{o})$ finds a sequence of operations that fulfils the constraints. Recall that operations include parameters and values, and that they too must be taken care of if some constraints restrict them. $P$ acts as the synthesis engine, while $\varphi$ acts as the requirements expressed in the language $\mathcal{L}$.

While there exist multiple ways of solving such a problem, a particularly appealing one is to use logic, or logic-based formalisms, to represent constraints. Rather than providing *ad hoc* algorithms or scripts tailored to a handful of situations, representing each constraint as an *assertion* that must be fulfilled by a potential solution opens the way to more general resolution algorithms. Using logic as the underlying language for expressing dependencies amounts to formulating self-* properties and web service composition broadly as a *constraint satisfaction problem*. Therefore, representing the configuration guidelines in a language $\mathcal{L}$ for which an algorithm $P$ exists allows us to leverage any available CSP tool to solve this problem.

The previous concepts translate easily into logical terminology. A solution $\bar{o}$ consistent with the constraints expressed by $\varphi$ is called a *model* of $\varphi$; we note that fact $\bar{o} \models \varphi$. If the set of constraints expressed by $\varphi$ admits a solution, $\varphi$ is said to be *satisfiable*. Procedure $P$ is called a *satisfiability solver*. It need not return a solution for every input. If, for any input $\varphi$, the solution $\bar{o}$ (if any) returned by $P$ is such that $\bar{o} \models \varphi$, $P$ is said to be *sound*. Conversely, if any input $\varphi$ for which there exists a solution $\bar{o}$ is found by $P$, $P$ is said to be *complete*. A procedure that is both sound and complete computes exactly all the solutions for inputs that have one.

A first advantage of such an approach is its genericity. As long as the language $\mathcal{L}$ is sufficient for expressing all necessary constraints, the procedure $P$ is not tied to any particular instance of a problem and acts as a general problem solver.

A second advantage is that logical formalisms have been thoroughly studied. For example, we shall see in Section 5 a theoretical result about first-order logic which tells us that the problem of finding a finite model for an arbitrary set of constraints is undecidable —that is, unless a tool uses a restricted form of first-order logic, its algorithm cannot be both sound and complete [28, 51]. This kind of global results are seldom available for *ad hoc* methods that do not rest on formal grounds.

Early work on that topic can be traced to Kautz [43], who launched the concept of "planning as satisfiability". However, the present approach differs from traditional planning in some important aspects:

– In planning, transition systems' "states" are atomic objects, and "actions" are represented as binary relations on the set of states. Intuitively, the execution of an action transforms one state into another. Automated sequence generation uses a simplified model where one is simply interested in actions; the internal state of the system is not taken into account, and the focus is on the generation of a sequence of actions satisfying a set of constraints.
– In planning, a "goal state" must eventually be reached. By the previous remark, there cannot be such a "goal" in automated sequence generation; a solution to an automated sequence generation problem is simply a sequence of actions that satisfies the constraints.
– Planning typically aims at computing a complete solution from beginning to end in a single step. Some applications of automated sequence generation are rather interactive (or incremental), as in automated stub generation.

## 3.3 Desirable Properties for a Logic

The above examples help us determine a few desirable properties that a candidate configuration logic must have in order to be helpful in the scenarios described previously.

First, we can distinguish three types of constraints that can be potentially expressed in the target formal language; Figure 3 provides an example of each.

### *3.3.1 1) Support for Static Constraints*

A first type of constraint consists of an interdependence relation between multiple data elements inside a single operation or state of the system: we call these dependencies static, which refer to one operation at one moment in time. This relation is described by the dashed line labelled “1” in Figure 3: two parameters in the structure of an operation are constrained by an arbitrary relation. For example, the context could require that parameters $p_l$ and $p_n$ within the same operation have identical (or distinct) values.

To handle this type of constraint, the chosen logic must express relations on parameter-value pairs, such as First-Order Logic [51], which will be described in Section 5.

### *3.3.2 2) Support for Temporal Constraints*

A second type of constraint consists of an interdependence relation on the sequence of two operations: the constraint is hence called dynamic, since it imposes a relation between two different moments in time. This relation is described by the dashed line labelled “2” in Figure 3; for example, the context could require that the execution of operation “C” always be preceded by the reception of a response labelled “B”.

To handle this type of constraint, the chosen logic must express temporal relations. It must include sequencing functions mirroring the temporal operators of logics like Linear Temporal Logic (LTL) [57], which will be described in Section 4.

### *3.3.3 3) Support for Data-Aware Constraints*

These constraints are the combination of the two previous types of constraints: a relation is imposed on the data content of two operations in two distinct moments in time, as shown by the dashed line labelled “3” in Figure 3. It cannot be considered as a simple static constraint either, since two distinct operations are involved. In the same way, it cannot be considered as a simple temporal constraint, since the content of operations is part of the constraint, and not only the sequencing of these operations.

In the VPN scenario, this is exemplified by Sequential Rule 2. Indeed, this constraint requires that the arguments of all `ip vrf forwarding` and `ip vrf` commands have the same value. This implies an access to the `as` parameter in each such command, and a comparison between values of any pair of them.

In the web service scenario, this is exemplified by Sequential Rule 4. By referring to “every buy and sell order”, this new constraint requires that each bill ID appearing in a confirmation eventually appears inside a cancellation or a cash transfer. It implies an access to the data content of the message (the bill ID), and moreover correlates data values at two different moments along the message trace.

This need to access and correlate the data content in multiple messages is not an exception and appears in many other natural properties. Data-aware constraints are therefore more than the sum of their parts; they are best expressed in a "hybrid" variant of both LTL and first-order logic such as LTL-FO$^+$, which will be described in Section 6. The reader is referred to [35] for a detailed description of data-awareness.

In addition to these requirements regarding expressiveness, the target candidate logic should have a number of other properties.

### *3.3.4 4) Decidability*

To be used as a general constraint solver, the candidate logic must have an algorithm $P$ as described above. This algorithm should be sound, and ideally complete. However, there exist logics for which no such algorithm exists (and will ever exist). Such logics are called *undecidable*. The undecidability of a logic has an important consequence for the task at hand. Suppose that the language used for describing constraints is undecidable. Then any algorithmic procedure $P$ dealing with such a language is *bound* to be imperfect. More precisely, either:

1. It is not sound (and can generate "wrong" solutions) or incomplete (there exist situations where a satisfying assignment exists but $P$ cannot find it); or
2. It cannot be guaranteed to terminate every time —that is, for some inputs, $P$ might fall into an infinite loop.

Note that this consequence stems not from the inability for a programmer to write a correct procedure, but rather from a formal property of the underlying specification language. Automated reasoners for semantic web languages, such as OWL, are often limited by the fact that their underlying language (first-order logic) is undecidable. Remark, however, that decidability can sometimes be restored by imposing restrictions on the way formulæ are written. The loosely guarded fragment of first-order logic, which will be presented in Section 6.2, is an example of conditions on formulæ that guarantees they are decidable.

### *3.3.5 5) Finite/small model*

Another consideration for the logics under study is the size of the solution that their satisfiability algorithm can return. For some logics, there exist sets of constraints whose solutions are infinite. Take for example the statement "for every element $x$, there exists an element $y$ such that $y > x$". Any model of this formula must have an infinite number of elements (an appropriate model for this expression would be the set of natural numbers, which is infinite).

Clearly, in the present context, one is not interested in solutions involving an infinite number of operations, or an infinite number of parameters for some operation. It is therefore desirable to readily identify sets of constraints leading to such infinite models, or to use logics that prevent the creation of such sets altogether. A logical language that guarantees that if a solution exists, then at least one of them has a finite cardinality, is said to have the *finite model property.* More interestingly, some logics have the stronger *small model property*, where the existence of a solution ensures that at least one of them is not only finite, but bounded by a function of the size of the original set of constraints.

#### *3.3.6 6) Existence of automated solvers*

Since the purpose of a logical modelling of sequential constraints is to tap into existing tools for generating models, preference should be given to languages where solvers are available.

In the next sections, we survey various formalisms that could be suitable candidates to express and compute sequences of operations. Each formalism is analyzed with respect to the criteria mentioned above. We shall see that some of them fair differently with respect to each criterion. Yet, any candidate logic for automated sequence generation would be worth studying under these various angles. However, it is not in the paper's scope to express a preference for any of them, as the choice of the "best" language involves a careful compromise between these aspects.

## 4 Linear Temporal Logic

Linear Temporal Logic [57] is a logic aimed at describing sequential properties along paths in a given Kripke structure. It traces its origins in hardware verification, where it has been widely used to express properties of sequential circuits, among other things [64].

### 4.1 Description

LTL's syntax starts with *ground terms*, which generally are symbols that can take the value true ($\top$) or false ($\bot$). Such symbols can be combined to form compound statements, with the use of the classical propositional operators $\vee$ ("or"), $\wedge$ ("and"), $\neg$ ("not") and $\rightarrow$ ("implies"). To express sequential relationships, *temporal operators* have been added.

The first such modal operator is **G**, which means "globally". Formally, the formula $\mathbf{G}\ \varphi$ is true on a given path $\pi$ when, for all states along this path, the formula $\varphi$ is true. The second modal operator commonly used is **X** ("next"). The formula $\mathbf{X}\ \varphi$ is true on a given path $\pi$ of the Kripke structure when the next state along $\pi$ satisfies $\varphi$. A formula of the form $\mathbf{F}\ \varphi$ ("eventually") is true on a path when at least one state of the path satisfies $\varphi$. Finally, the operator **U** ("until") relates two expressions $\varphi$ and $\psi$; the formula $\varphi\,\mathbf{U}\,\psi$ states that 1) $\psi$ eventually holds, and 2) in the meantime, $\varphi$ must hold in every state. More details can be found in [17].

We must first define the ground terms for our theory:

1. From the set $A$ of actions, we define an equal number of *action terms* $\alpha_a$ for every $a \in A$. On a specific operation $o = (a', \star)$, the term $\alpha_a$ is true exactly when $a = a'$.
2. For every pair of parameter $p \in P$ and value $v \in V$, we create a *tuple term* $t_{p,v}$. On a specific operation $o = (a', \star)$, the term $t_{p,v}$ is true exactly when $v \in \star(p)$.

Equipped with these ground terms, it is first possible to express the schema and domain functions $S$ and $D$ defined in Table 1. For example, since $S(\texttt{ip vrf}) = \{\texttt{vrf-name}\}$, then for any operation whose action is `ip vrf`, only the parameter vrf-name is allowed to have some value. This can be written as:

$$\bigwedge_{p \in P \setminus \{\texttt{ip vrf}\}} \bigwedge_{v \in V} \mathbf{G}\,(\alpha_{\texttt{ip vrf}} \rightarrow \neg t_{p,v})$$

The symbol $\bigwedge_{v \in V}$ is a "meta-symbol", not part of the logic itself. It means that the formula that follows must be repeated once for every value in the set $V$, by replacing any occurrence of $v$ by that value. For example, if $V = \{k_1, k_2, k_3\}$, then $\bigwedge_{v \in V} t_{p,v}$ would be expanded into $t_{p,k_1} \wedge t_{p,k_2} \wedge t_{p,k_3}$. These meta-symbols can be nested, such that the preceding formula amounts the the repeated conjunction of $\mathbf{G}\,(\alpha_{\texttt{ip vrf}} \rightarrow \neg t_{p,v})$ for all $p \in P \setminus \{\texttt{ip vrf}\}$ and $v \in V$. Hence the previous expression represents $|V|(|P| - 1)$ "copies" of $\mathbf{G}\,(\dots)$, separated by the $\wedge$ connective.

The formula states that every operation in the trace is such that, if the operation's action is `ip vrf`, then every tuple term $t_{p,v}$ must be false whenever $p$ is not `ip vrf`. Such an expression must be repeated for every action and its corresponding schema; this can be formalized generally as follows:

$$\bigwedge_{a \in A} \bigwedge_{p \in P \setminus S(a)} \bigwedge_{v \in V} \mathbf{G}\ (\alpha_a \rightarrow \neg t_{p,v}) \qquad (1)$$

In the same way, domains for each parameter can be formalized by a similar LTL formula:

$$\bigwedge_{p \in P} \bigwedge_{v \in V \setminus D(p)} \mathbf{G}\,\neg t_{p,v} \qquad (2)$$

Sequential constraints in Section 2.1 can now be also expressed as LTL formulæ. For example, Sequential Constraint 1 becomes the following LTL expression:

**LTL Sequential Formula 1**

$$\neg\alpha_{\texttt{rd}}\,\mathbf{U}\,\alpha_{\texttt{ip vrf}} \qquad (3)$$

This formula states that on any trace of operations $\bar{o} = o_0, o_1, \dots,$ action term $\alpha_{\texttt{rd}}$ is not true on any message before action term $\alpha_{\texttt{ip vrf}}$ becomes true. In other words, the action of an operation cannot be `rd` before some operation has `ip vrf` as its action; this is indeed equivalent to Sequential Constraint 1.

Similarly, Sequential Formula 2 becomes a slightly more complex expression:

**LTL Sequential Formula 2**

$$\mathbf{G}\left(\alpha_{\texttt{ip vrf}} \rightarrow \bigwedge_{v \in V} \left(t_{\texttt{as},v} \rightarrow \mathbf{G}\left(\alpha_{\texttt{ip vrf forwarding}} \rightarrow t_{\texttt{as},v}\right)\right)\right) \tag{4}$$

The subformula $\mathbf{G}\,(\alpha_{\texttt{ip vrf forwarding}} \rightarrow t_{\texttt{as},v})$ indicates that, at any point in the trace, any operation with action `ip vrf forwarding` has its `as` parameter set to some value $v$. The expression $\bigwedge_{v \in V}(\cdot)$ indicates to take the conjunction of the expression inside the parentheses successively for every value $v \in V$. As a whole, the formula expresses the fact that at any point in the trace, an operation with action `ip vrf` entails that, for any value $v$ that holds for its parameter `as`, globally any occurrence of `ip vrf forwarding` has that same value $v$ for its parameter `as`.

## 4.2 Analysis

The conjunction of formulæ (1)–(4) is a (long) LTL formula that provides a complete specification $\varphi$ of the VRF configuration constraints, expressed in Linear Temporal Logic. It follows that any trace of operations $\bar{o}$ such that $\bar{o} \models_{\text{LTL}} \varphi$ will be a valid sequence of operations, with respect to these requirements.

We now revisit the desirable properties enumerated in Section 3.3, with respect to LTL. First, one can see that, while temporal constraints are easily expressed with LTL's operators, static and data-aware constraints must be simulated through the repeated enumeration of formulæ, changing a Boolean variable each time. Hence, static and data-aware properties become long repetitions of seemingly similar copies of the same formula; more precisely, if $|V|$ is the domain size for values and $k$ is the number of "meta-symbols" ($\bigvee$ or $\bigwedge$) ranging over $V$, then the length of the resulting LTL formula is in $O(|V|^k)$. Moreover, the expression of that formula must be changed every time the data domain changes.

LTL satisfiability is decidable, and its complexity is PSPACE-complete [26]. As a matter of fact, any LTL formula can be translated into an equivalent *Büchi automaton* [19,26] whose size is bounded; hence LTL also has the small model property.

While there exist satisfiability algorithms for LTL, which generally involve translating a formula into an equivalent [19, 26], it is also possible to put existing software, called LTL model checkers, to good use. A *model checker* takes as input a finite state machine $K$, called a Kripke structure, and an LTL specification $\varphi$. It then exhaustively checks that any possible execution of $K$ satisfies $\varphi$. Popular model checkers for LTL include SPIN [39], NuSMV [15], and Spot [24].

As a by-product of their exhaustive verification, most model checkers produce counter-example traces of a formula, in the case where $K$ does not satisfy $\varphi$. It is possible to benefit from this counter-example generation mechanism to find a sequence that does not violate any constraint. It suffices to submit the formula $\neg\varphi$ for verification. If there exists a counter-example for $\varphi$, then such a trace must necessarily satisfy $\varphi$, giving by the same occasion a valid deployment sequence. It suffices to give the model checker a Kripke structure $K$ where all states are linked to each other, called a universal Kripke structure, and the negation of the constraints as the specification to verify. Such a technique was suggested by Rozier and Vardi [61] and analyzed empirically for a variety of model checkers.

## 5 First-Order Logic

In Linear Temporal Logic, events are considered as "atomic": they do not contain parameters. To simulate parameters in events, one must therefore resort to create one propositional variable for every parameter and every value, and correctly assign truth values to these variables in every possible operation. Constraints on values become tedious to write, since the same property must be enumerated using every possible value. It would be desirable to have a mechanism where data inside events could be referred to, memorized, and compared at a later time. First-order logic is a formalism that provides such a mechanism.

### 5.1 Description

In first-order logic (FOL), the traditional propositional connectives $\vee$, $\wedge$ and $\neg$ are used to connect *predicates* $p_1, p_2, \dots$, which take their values in some arbitrary domain $\mathcal{D}$, and return either $\top$ or $\bot$. The *arity* of predicates is the number of arguments they take; for example, the binary predicate $p(x, y)$ is a function from $\mathcal{D} \times \mathcal{D}$ to $\{\top, \bot\}$.

These predicates and connectives are complemented with a set of two quantifiers. The expression $\exists x : \varphi(x)$ states that there exists some element $d \in \mathcal{D}$ such that $\varphi$ is true when all occurrences of $x$ are replaced by $d$. Similarly, $\forall x : \varphi(x)$ asserts the same property, but for all elements $d \in \mathcal{D}$.

In counterpart, the temporal operators and intervals that were available in LTL have disappeared. Temporal relationships between events must therefore be simulated, using additional variables or relations. More specifically, a trace of messages can be represented by a set of three ground predicates:

1. From the set $A$ of actions, we define an *action predicate* $\alpha : \mathbb{N} \times A \rightarrow \{\top, \bot\}$. On a specific operation trace $\bar{o} = o_1, o_2, \ldots$, for any $i \in \mathbb{N}$ and $a \in A$, the predicate $\alpha(i, a)$ is true exactly when $o_i = (a, \star)$.
2. From the sets of parameters $P$ and values $V$, we create a *tuple predicate* $t : \mathbb{N} \times P \times V \rightarrow \{\top, \bot\}$. On a specific operation $o_i = (a, \star)$ in an operation trace, for any $p \in P$ and $v \in V$, $t(i, p, v)$ is true exactly when $v \in \star(p)$.

The quantifiers are handy for representing the schema and domain functions. The constraint on the `ip vrf` command, for example, becomes the following first-order formula:

$$\forall p \in P \setminus \{\texttt{ip vrf}\} : \forall v \in V : \forall i \in \mathbb{N} : \alpha(i, \texttt{ip vrf}) \rightarrow \neg t(i, p, v)$$

This formula states that, for every pair of parameters (except `ip vrf`) and values $p$ and $v$, and for every index $i$, if the action of the $i$-th operation of the trace is `ip vrf`, then no other parameter-value tuple can be present in the message.

More generally, this expression can be repeated for every action and its corresponding schema, as follows:

$$\forall a \in A : \forall p \in P \setminus S(a) : \forall v \in V : \forall i \in \mathbb{N} : \alpha(i, a) \rightarrow \neg t(i, p, v) \qquad (5)$$

In the same way, domains for each parameter can be formalized by a similar first-order formula:

$$\forall p \in P : \forall v \in V \setminus D(p) : \forall i \in \mathbb{N} : \neg t(i, p, v) \qquad (6)$$

The sequential constraints in Section 2.1 can now be expressed as first-order formulæ. However, special care must be taken to correctly express sequential dependencies as appropriate constraints on the operation indices.

**First-Order Sequential Formula 1**

$$\exists i \in \mathbb{N} : (\alpha(i, \texttt{ip vrf}) \wedge \forall j \in \mathbb{N} : (j < i \rightarrow \neg \alpha(j, \texttt{rd}))) \qquad (7)$$

This formula states that in the operation trace $\bar{o} = o_1, o_2, \ldots$, there exists some operation $o_i$ whose action is `ip vrf`, and such that for every *preceding* operation $o_j$ (where $j < i$), the action for message $o_j$ is not `rd`. This is indeed equivalent to saying that `ip vrf` must precede `rd` in the operation trace.

Similarly, Sequential Constraint 2 can be formalized as follows:

**First-Order Sequential Formula 2**

$$\forall i \in \mathbb{N} : \forall j \in \mathbb{N} : \forall v \in V : (\alpha(i, \texttt{ip vrf}) \wedge \alpha(j, \texttt{ip vrf forwarding})) \rightarrow (t(i, \texttt{as}, v) \leftrightarrow t(j, \texttt{as}, v)) \qquad (8)$$

This formula states that for any two operations $o_i$ and $o_j$ in a trace, if $o_i$ has action `ip vrf` and $o_j$ has action `ip vrf forwarding`, then their `as` parameter is equal. The $\leftrightarrow$ operator means "if and only if", and returns true when both its left- and right-hand members have the same truth value. In the present case, this means that for every value $v \in V$, either $o_i$ and $o_j$ both have $v$ as the `as` parameter, or neither does. The global formula is indeed equivalent to stating that the `as` parameter of all `ip vrf` and `ip vrf forwarding` commands is the same for all their occurrences in a trace.

## 5.2 Analysis

The conjunction of formulæ (5)-(8) is a first-order formula that provides a complete specification of the VRF constraints expressed in first-order logic. Again, any trace of operations $\bar{o}$ such that $\bar{o} \models_{\text{FOL}} \varphi$ will be a valid sequence of operations with respect to these requirements. A model of $\varphi$ therefore consists in the complete definition of predicates $\alpha$ and $t$ for every $i \in \mathbb{N}$, $a \in A$, $p \in P$ and $v \in V$.

The results of the analysis of first-order logic with the criteria presented in Section 3.3 are almost the opposite as those obtained for LTL in the previous section. First-order logic allows a succinct description of static constraints through the use of universal and existential quantifiers; however, classical FOL does not provide the equivalent of LTL temporal operators. Those must be simulated by explicitly giving constraints on the indices of operations in a trace, stating for which indices a particular formula must hold. This makes the temporal relations much less explicit, as First-Order Sequential Formula 2, for example, shows. Hence, data-aware constraints are also tedious to write, but for a different reason than in LTL.

First-order logic is strictly richer than linear temporal logic: a classical result shows that first-order logic encompasses all of LTL. However, this richness comes at the price of complexity. A result by Trakhtenbrot [63] states that for a first-order language including a relation symbol that is not unary, satisfiability over finite structures is undecidable.

We have seen in Section 3.3 the consequences of undecidability for the constraint language. However, adding constraints to the structure of first-order formulæ might restore decidability. In the particular case where all domains are considered finite and are known in advance, first-order logic statements become decidable [46]. Hence there exist "model finders" for first-order logic that work in that particular case.

Some of the most well-known first-order model finders are Mace [50], Paradox [16], and SEM [65]; they all assume that the cardinality of all domains is bounded. This has an important consequence: it entails that even the length of the trace must be fixed in advance. Hence, in First-Order Sequential Formula 1, the quantifier $\exists i \in \mathbb{N}$ actually becomes $\exists i \in \{0, \ldots, n\}$ for some constant $n$.

A book by Börger, Grädel, and Gurevich [12] contains an extensive survey on results that assert that certain fragments of first order-logic have a decidable satisfiability problem or not.

## 6 Linear Temporal Logic with First-Order Quantification

The specification, efficient validation and satisfiability of data-aware constraints is currently an open problem. Although there exists a number of adequate formalisms to specify static and temporal constraints, by the previous observations, they cannot simply be used "side by side" to cover the case of hybrid constraints.

For example, while Linear Temporal Logic allows a succinct formalization of temporal relations, its lack of a quantification mechanism makes the expression of data constraints cumbersome: the same formula must be repeated for every combination of values, replacing them by the appropriate ground terms each time. This results in a potentially exponential blow-up of the original specification, in terms of the domain size.

In contrast, first-order logic's quantification mechanism allows to succinctly express relationships between data parameters between events, but its lack of temporal operators makes the expression of sequential relationships much less natural.

One is therefore interested in striking a balance between the two extremes that these logics provide.

### 6.1 Description

Linear Temporal Logic with First-Order quantification, called LTL-FO$^+$, aims at this middle ground [33]. As its name implies, LTL-FO$^+$ is a logic where first-order quantifiers and LTL temporal operators can be freely mixed. Its basic building blocks are ground predicates defined as follows:

- From the set $A$ of actions, we define an *action predicate* $\alpha : A \to \{\top, \bot\}$. On a specific operation $o$, $\alpha(a)$ is true exactly when $o = (a, \star)$.
- From the sets of parameters $P$ and values $V$, we create a *tuple predicate* $t : P \times V \to \{\top, \bot\}$. On a specific operation $o = (a, \star)$, for any $p \in P$ and $v \in V$, $t(p, v)$ is true exactly when $v \in \star(p)$.

One remarks that the predicates are similar to first-order logic, except that they no longer require the "index" specifying at what particular operation in a trace they refer. This, similarly to linear temporal logic, is handled implicitly by the temporal operators.

The Boolean connectives and LTL temporal operators carry their usual meaning. First-order quantifiers, on the other hand, are a limited version of their respective First-Order version. In LTL-FO$^+$, each quantifier is of the form $\exists_p x : \varphi$ or $\forall_p x : \varphi$ and has a subscript, $p$, which determines which values are admissible for $x$. More precisely, $\forall_p x : \varphi$ is true for some operation $o = (a, \star)$ if and only if for all values $c \in \star(p)$, $\varphi$ holds when $x$ is replaced by $c$. Similarly, $\exists_p x : \varphi$ holds if some value $c \in \star(p)$ is such that $\varphi$ holds when $x$ is replaced by $c$. The choice of appropriate values for $x$ therefore depends on the current operation pointed to in the trace; this, in turn, is modulated by the LTL temporal operators.

The development of such a "hybrid" between FOL and LTL has led to many variants of this strategy; notable proponents include LTL-FO [22], EQCTL [45], QCTL [59], Eagle [10], FOTLX [23], and RuleR [11]. However, the particular quantification mechanism presented here is unique to LTL-FO$^+$.

Schema and domain constraints are represented in the same way as for LTL, as is Sequential Constraint 1:

**LTL-FO$^+$ Sequential Formula 1**

$$\neg\alpha(\texttt{ip vrf})\, \mathbf{U}\, \alpha(\texttt{rd}) \tag{9}$$

However, using the LTL-FO$^+$ quantifiers, one can represent Sequential Constraint 2 in a much shorter way:

**LTL-FO$^+$ Sequential Formula 2**

$$\mathbf{G}\ (\alpha(\texttt{ip vrf}) \to \forall_{\texttt{as}} v : \mathbf{G}\ (\alpha(\texttt{ip vrf forwarding}) \to t(\texttt{as}, v))) \tag{10}$$

## 6.2 Analysis

The conjunction of formulæ (1)-(10) is an LTL-FO$^+$ formula that provides a complete specification of the VRF constraints. Again, any trace of operations $\bar{o}$ such that $\bar{o} \models_{\text{LTL-FO}^+} \varphi$ will be a valid sequence of operations with respect to these requirements.

As one can see, LTL-FO$^+$ is the formalism where Sequential Formulæ have the shortest expression, compared to both LTL and FOL. Static constraints become similar to first-order logic, while temporal constraints can use the LTL operators. Obviously, data-aware constraints can use both.

However, there is currently no dedicated *satisfiability* solver available for LTL-FO$^+$. Actually, at first glance it is not even clear that LTL-FO$^+$ is decidable, since it includes a form of first-order quantification. However, one can demonstrate that the quantification mechanism of LTL-FO$^+$ is actually similar to the *loosely guarded fragment* of first-order logic [38]. Guarded logic is a generalization of modal logic in which all quantifiers must be relativized by atomic formulæ [7]. If we let $\bar{x}$ and $\bar{y}$ be tuples of variables, quantifiers in the guarded fragment (GF) of first-order logic appear only in the form:

$$\exists \bar{y}\, (\gamma(\bar{x}, \bar{y}) \wedge \psi(\bar{x}, \bar{y}))$$

$$\forall \bar{y}\, (\gamma(\bar{x}, \bar{y}) \to \psi(\bar{x}, \bar{y}))$$

The predicate $\gamma$, called the *guard*, must contain all free variables of $\psi$. In this context, it suffices to define the guard as $\gamma(p, x) \equiv \star(p) = y$ and rewrite all LTL-FO$^+$ formulæ as guarded LTL formulæ with general first-order quantifiers:

$$\exists_p x : \varphi \equiv \exists x : (\gamma(x, p) \wedge \varphi)$$
$$\forall_p x : \varphi \equiv \forall x : (\gamma(x, p) \rightarrow \varphi)$$

The advantage of such a rewriting is that the loosely guarded fragment of first-order logic is decidable [27] and has the finite-model property [38]; hence LTL-FO$^+$ is also decidable.

Obviously, LTL-FO$^+$ can be translated back into either LTL or FOL expressions and handled by their respective solvers. Experiments on this concept have led to a first attempt at LTL-FO$^+$ satisfiability solving [29,30]. This raises the concept that the language used for specification can be different from the underlying representation used by the automated sequence generator.

## 7 Conclusion

In this paper, we have shown how sequential constraints arise naturally in a variety of computing scenarios, including the management of network devices and the interaction between web services. The automated generation of sequences of operations, following these constraints, amounts to various concepts depending on the field over which it applies. For example, generating sequences of operations corresponds to self-configuration when applied to the management of network device configurations.

The paper argued that such properties be expressed in a declarative manner by means of a formal language, in particular a *logic*. In such a case, the automated generation of sequences becomes a simple instance of *satisfiability solving*, for which solvers already exist in a number of cases.

We first concentrated on two natural candidates for the task at hand, namely Linear Temporal Logic and First-Order Logic. However, we have shown that many of these constraints correlate sequences of operations and parameters inside these operations in such a way that both types of properties cannot be expressed in isolation. This called for the presentation of a family of hybrid formalisms for expressing "data-aware" constraints, a notable proponent being LTL-FO$^+$.

All three logics have been compared with respect to a number of criteria relevant to automated sequence generation. A summary of these results can be found in Table 2.

The paper focused on a linear temporal–first order characterization of sequential constraints. However, the authors are fully aware that there exist other formalisms that could be suitable for this task. For example, Allen's Interval Temporal Logic [5], Temporal Description Logics [47], interface grammars [32]

| | LTL | FOL | LTL-FO$^+$ |
|---|---|---|---|
| Static properties | + | ++ | ++ |
| Temporal properties | ++ | + | ++ |
| Data-aware properties | + | + | ++ |
| Decidable | Yes | No* | Yes |
| Small model | Yes | No* | Yes |
| Existing solvers | + | ++ | – |

**Table 2** A summary of the results for the logics studied in this paper.

are all formalisms that should be studied in their own right with respect to the criteria specified in this paper. The presence of such a large roster of possibilities reveals the extent of the uncharted territory on the topic of automated sequence generation with data, as well as its potential for furthering the reach of declarative specifications in computing.

## References

1. *Proceedings of the 20th Conference on Systems Administration (LISA 2006), Washington, D.C., USA, December 3-8, 2006.* USENIX, 2006.
2. Amazon e-commerce service, 2009. http://docs.amazonwebservices.com/-AWSEcommerceService/2005-03-23/.
3. soapUI: the web services testing tool, 2009. http://www.soapui.org/.
4. Cisco IOS switching services command reference, release 12.3, 2010. http://-www.cisco.com/en/US/docs/ios/12_3/switch/command/reference/swtch_r.html.
5. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
6. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services, Concepts, Architectures and Applications.* Springer, 2004.
7. H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, (27):217–274, 1998.
8. T. R. Arnold II. *Visual Test 6 Bible.* Wiley, 1998.
9. L. Baresi, C.-H. Chi, and J. Suzuki, editors. *Service-Oriented Computing, 7th International Joint Conference, ICSOC-ServiceWave 2009, Stockholm, Sweden, November 24-27, 2009. Proceedings*, volume 5900 of *Lecture Notes in Computer Science*, 2009.
10. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.
11. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. *Journal of Logic and Computation*, 2008.
12. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer, 1997.
13. M. Burgess and A. Couch. Modeling next generation configuration management tools. In *LISA* [1], pages 131–147.
14. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1, W3C note, 2001.
15. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.
16. K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.
17. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 2000.

18. F. Daniel and B. Pernici. Insights into web service orchestration and choreography. *Int. Journal of E-Business Research*, 2(1):58–77, 2006.
19. M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In N. Halbwachs and D. Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 1999.
20. S. Demri and C. S. Jensen, editors. *15th International Symposium on Temporal Representation and Reasoning, TIME 2008, Université du Québec à Monteéal, Canada, 16-18 June 2008*. IEEE Computer Society, 2008.
21. N. Desai, R. Bradshaw, and C. Lueninghoener. Directing change using Bcfg2. In *LISA* [1], pages 215–220.
22. A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web services. In A. Deutsch, editor, *PODS*, pages 71–82. ACM, 2004.
23. C. Dixon, M. Fisher, B. Konev, and A. Lisitsa. Practical first-order temporal reasoning. In Demri and Jensen [20], pages 156–163.
24. A. Duret-Lutz and D. Poitrenaud. Spot: An extensible model checking library using transition-based generalized büchi automata. In D. DeGroot, P. G. Harrison, H. A. G. Wijshoff, and Z. Segall, editors, *MASCOTS*, pages 76–83. IEEE Computer Society, 2004.
25. D. Gaïti, G. Pujolle, M. Salaün, and H. Zimmermann. Autonomous network equipments. In I. Stavrakakis and M. Smirnov, editors, *WAC*, volume 3854 of *Lecture Notes in Computer Science*, pages 177–185. Springer, 2005.
26. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.
27. E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64(1):1719–1742, March 1999.
28. E. Grädel, P. G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Y. Vardi, Y. Venema, and S. Weinstein. *Finite Model Theory and Its Applications.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2007.
29. S. Hallé. Automated generation of web service stubs using ltl satisfiability solving. In M. Bravetti and T. Bultan, editors, *WS-FM*, volume 6551 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 2010.
30. S. Hallé. Causality in message-based contract violations: A temporal logic "whodunit". In *EDOC*, pages 171–180. IEEE Computer Society, 2011.
31. S. Hallé, R. Deca, O. Cherkaoui, R. Villemaire, and D. Puche. A formal validation model for the Netconf protocol. In A. Sahai and F. Wu, editors, *DSOM*, volume 3278 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 2004.
32. S. Hallé, G. Hughes, T. Bultan, and M. Alkhalaf. Generating interface grammars from WSDL for automated verification of web services. In Baresi et al. [9], pages 516–530.
33. S. Hallé and R. Villemaire. Runtime monitoring of message-based workflows with data. In *EDOC*, pages 63–72. IEEE Computer Society, 2008.
34. S. Hallé and R. Villemaire. XML methods for validation of temporal properties on message traces with data. In R. Meersman and Z. Tari, editors, *CoopIS/DOA/ODBASE*, volume 5331 of *Lecture Notes in Computer Science*, pages 337–353. Springer, 2008.
35. S. Hallé, R. Villemaire, and O. Cherkaoui. Specifying and validating data-aware temporal web service properties. *IEEE Trans. Software Eng.*, 35(5):669–683, 2009.
36. S. Hallé, R. Villemaire, and O. Cherkaoui. Logical methods for self-configuration of network devices. *Formal and Practical Aspects of Autonomic Computing and Networking: Specification, Development and Verification*, pages 189–216, 2011.
37. P. Hinnelund. Autonomic computing. Master's thesis, School of Computer Science and Engineering, Royal Institute of Engineering, March 2004.
38. I. M. Hodkinson. Loosely guarded fragment of first-order logic has the finite model property. *Studia Logica*, 70:205–240, 2002.
39. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.
40. G. Hughes, T. Bultan, and M. Alkhalaf. Client and server verification for web services using interface grammars. In T. Bultan and T. Xie, editors, *TAV-WEB*, pages 40–46. ACM, 2008.

41. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *ICSE*, pages 730–733, 2000.
42. J. Josephraj. Web services choreography in practice, 2005. http://www-128.ibm.com/developerworks/webservices/library/ws-choreography/.
43. H. A. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.
44. S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal*, 11(4), 2004.
45. O. Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. In P. Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 325–338. Springer, 1995.
46. L. Löwenheim. Uber möglichkeiten im relativkalkül. *Math. Annalen*, 76:447–470, 1915.
47. C. Lutz, F. Wolter, and M. Zakharyaschev. Temporal description logics: A survey. In Demri and Jensen [20], pages 3–14.
48. D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic markup for web services, 2008. http://www.ai.sri.com/daml/-services/owl-s/1.2/overview.
49. E. Martin, S. Basu, and T. Xie. Automated testing and response analysis of web services. In *ICWS*, pages 647–654. IEEE Computer Society, 2007.
50. W. McCune. A davis-putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, 1994.
51. E. Mendelson. *Introduction to Mathematical Logic, Fourth Edition.* Springer, 1997.
52. S. Narain. Towards a foundation for building distributed systems via configuration, 2004. Retrieved February 11th, 2010.
53. S. Narain. Network configuration management via model finding. In *LISA*, pages 155–168. USENIX, 2005.
54. S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, pages 77–88, 2002.
55. M. Parashar and S. Hariri. Autonomic computing: An overview. In J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *UPP*, volume 3566 of *Lecture Notes in Computer Science*, pages 257–269. Springer, 2004.
56. I. Pepelnjak and J. Guichard. *MPLS VPN Architectures.* Cisco Press, 2001.
57. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.
58. J. Rao and X. Su. A survey of automated web service composition methods. In J. Cardoso and A. P. Sheth, editors, *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2004.
59. A. Rensink. Model checking quantified computation tree logic. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2006.
60. E. C. Rosen and Y. Rekhter. BGP/MPLS VPNs (RFC 2547), March 1999.
61. K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In D. Bosnacki and S. Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 149–167. Springer, 2007.
62. A. S. Tanenbaum. *Computer Networks, 4th Edition.* Prentice Hall, 2002.
63. B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii nauk SSSR*, (70):569–572, 1950.
64. G. von Bochmann. Hardware specification with temporal logic: En example. *IEEE Trans. Computers*, 31(3):223–231, 1982.
65. J. Zhang and H. Zhang. System description: Generating models by SEM. In M. A. McRobbie and J. K. Slaney, editors, *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 308–312. Springer, 1996.