Towards the formal verification of data-intensive applications through metric temporal logic

Francesco Marconi[1], Marcello M. Bersani[1],
Madalina Erascu[2], and Matteo Rossi[1]

[1] DEIB, Politecnico di Milano, Milan, Italy
{francesco.marconi,marcellomaria.bersani,matteo.rossi}@polimi.it
[2] Institute e-Austria Timisoara & West University of Timisoara, Timisoara, Romania
merascu@info.uvt.ro

Abstract We present an approach for the automated formal verification of distributed systems based on the Storm technology. The approach is based on a formal model of the behavior of Storm topologies given in terms of the CLTLoc metric temporal logic extended with counters. We present a tool-supported mechanism to automatically generate formal models from high-level description of Storm topologies. The Zot formal verification tool is then used to check whether some desired properties hold for the modeled system or not. The analyzed properties concern the growth of the queues of the nodes of the Storm topology. Some experiments performed on example topologies show how the timing features of the modeled system influence the behavior of the queues of the nodes.

Keywords: Data-intensive Applications, Distributed Systems, Formal Verification, Storm Technology, Metric Temporal Logic

1 Introduction

Big Data is a prominent area, involving both academia and industry, researching innovative solutions to support the entire life-cycle (from design to deployment) of so-called data-intensive applications (DIAs), which are able to process huge amounts of information. Hence, defining frameworks for the development of DIAs that leverage Big Data technologies is nowadays of major importance.

The DICE project [9] defines techniques and tools for the data-aware quality-driven development of DIAs. In the DICE approach, designers model DIAs through UML diagrams tagged with suitable annotations capturing the features of Big Data applications, and in particular their *topology*. A topology provides an abstract representation of a DIA through directed graphs, where nodes are of two kinds: *computational nodes* implement the logic of the application by elaborating information and producing an outcome, whereas *input nodes* bring information into the application from the environment.

The semantics underlying the topology typically changes depending on the target Big Data technology. In this paper we focus on the Apache Storm [1]

technology—in which computational nodes are called *bolts*, and input nodes are called *spouts*—a framework which is widely used in applications that need reliable processing of unbounded streams of data, e.g. Groupon (www.groupon.com), The Weather Channel (www.weather.com), Spotify (www.spotify.com), etc. In Apache Storm applications, one of the key concerns is that time-related parameters such as emission rates of data do not induce an excessive load on the topology by accumulating data in nodes' queues. The latest version of the framework offers options to adapt these parameters at run-time (e.g., by slowing down the input nodes) to mitigate the issue, but this might negatively and unpredictably impact other features of the application. Hence, one would like to design the topology from the beginning in a way that run-time adaptation is not necessary.

In this paper, we approach such design with three contributions.

We define a formal model of DIAs based on the Storm technology. This model, which we call the *timed counter networks* model, is expressed through the Constraint LTL over clocks (CLTLoc) [7] metric temporal logic enriched with positive counters. CLTLoc allows users to express time delays, and the addition of positive counters allows for the description of memory usage issues such the evolution of the length of nodes' queues.

We allow for the automated verification of such formal models through the **D-VerT** (DICE Verification Tool) prototype tool. By performing formal verification tasks through **D-VerT**, designers can detect bad configurations producing undesired consequences, such as data processing delays causing an unbounded use of memory.

We define sufficient conditions for guaranteeing the soundness of the verification results obtained through **D-VerT**. In fact, the extension of CLTLoc with unbounded counters makes the logic undecidable in general, so we must guarantee that the conditions and abstractions introduced to make the verification technique applicable in practice do not generate spurious results.

The rest of the paper is structured as follows. Section 2 presents some related works and Section 3 gives an overview of the Apache Storm technology. Section 4 introduces CLTLoc extended with counters, and a sufficient condition guaranteeing the soundness of its satisfiability checking procedure. Section 5 introduces the formal model of Storm topologies, and Section 6 describes some experimental results carried out with the **D-VerT** tool. Section 7 concludes.

2 Related works

Formal verification of distributed systems has been the focus of several decades of software engineering research. Challenging tasks in this context are: (i) finding the right abstraction for the formal model of the real world (*formalization*); (ii) developing techniques to prove the correctness of the modeled systems (*verification*); and (iii) bridging the gap between formalization and verification, since the formal model is often too complex to be tackled by the verification methods. Various approaches exist for the formalization of distributed systems; however, to the best of our knowledge none focuses on Storm-like streaming technologies.

Timed counter networks, the novel model of Storm topologies introduced in this paper, are inspired from *vector addition systems with states* (VASS) [14] and Timed Petri Nets [13]. VASS are a subclass of *counter systems*; that is, they are finite-state automata augmented with counters, whose values are non-negative integers, and which can be incremented and decremented. VASS are also equivalent to Petri nets for decision problems such as boundedness, covering and reachability [15]. Since distributed systems have unreliable communication, timed counter networks are also similar to lossy VASS [8], an abstraction of FIFO-channel systems, when only the number of messages is relevant, but not their ordering. Unlike (lossy) VASS, timed counter networks can express timing constraints along system executions through the notion of clocks.

Timed counter networks are inherently non-deterministic, and their behavior is effectively captured through formalisms such as the counter-augmented CLTLoc. At first glance they also seem expressible in terms of formalisms such as Timed Petri Nets (TPN) [13]. However, CLTLoc is more suitable to this end because, typically, TPN-based models adopt, both in theory and in practice, an *urgent* semantics for the firing of transitions [4], where an enabled transition *must* fire when it reaches its upper time bound if it is not disabled earlier. This makes modeling the *possible* occurrence of events in timed counter networks (e.g., failures in Storm topologies) less natural. Moreover, the typical semantics of the firings of transitions in TPNs does not allow for the modeling of a policy such as the following: *dequeuing always removes the maximum number of available elements in the queue, but never more than k elements at the same time.* The model in Section 5, instead, makes use of this abstraction to represent the behavior of a node when it extracts new elements from its queue to process them.

Concerning formal verification issues, the reachability problem is decidable for lossy unbounded FIFO-channel models [3,12] which implies the decidability of the verification problem of safety properties for lossy VASS. To the best of our knowledge, lossy VASS have been investigated only from a theoretical point of view, and no verification tools handling them currently exist.

3 Overview of Apache Storm

Apache Storm [1] is a stream processing system that allows parallel, distributed, real-time processing of large-scale streaming data on horizontally scalable systems.

The key concepts in Storm applications are *streams* and *topologies*. Streams are infinite sequences of tuples that are processed by the application. Topologies are directed graphs of computation, whose nodes correspond to the operations performed over the data flowing through the application, and whose edges indicate how such operations are combined, i. e., the streaming paths between nodes.

There are two kinds of nodes, *spouts* and *bolts* (in the following also referred to as *topology components*). Spouts are stream sources. They generally get data from external systems such as queuing brokers (e. g., Kafka, RabbitMQ, Kestrel) or from other data sources, e. g., Twitter Streaming APIs. Bolts apply transformations over the incoming data streams and generate new output streams

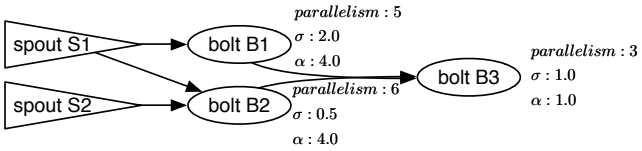

Figure 1: Example of Storm topology. Parameters σ and α are described in Sec. 5.

to be processed by the connected bolts. When a topology component generates new data into an output stream, it is said to *emit* tuples. Connections are defined at design time by the subscription of the bolts to other spouts or bolts. Fig. 1 shows an example of Storm topology that will be used in Section 6.

Spouts can be reliable or unreliable. The former keep track of all the tuples they emit, and if one of them fails to be processed by the entire topology within a certain timeout, then the spout re-emits it into the topology. The latter, instead, always emit each tuple only once, without checking for successful processing. Single bolts usually perform simple operations, such as filtering, join, functions, database interaction, which are combined in the topology to apply more complex transformations. IRichBolt and IRichSpout are the main Java interfaces to use for implementing the components of a topology. `execute()` is the method of IRichBolt defining the functionality of bolts; it reads the input tuples, processes the data, and emits (via the `emit()` method) the transformed tuples on the output streams. When the spouts are reliable, bolts have to acknowledge the successful or failed processing of each tuple at the end of the execution.

The Storm runtime is designed to leverage the computational power of distributed clusters. At a high level, its architecture is composed of one *master node*, and several *worker nodes*. One or more *worker processes* can be instantiated on a worker node, each of them executing different parts of the same topology. Each worker process runs a JVM where one or more *executors* (i.e. threads) are spawned. Executors can run one or more *tasks* which, in turn, can execute a spout or a bolt. The configuration of the topology defines the number of worker processes and, for each component (spout or bolt), the number of executors running it in parallel (the value of *parallelism* in Fig. 1) and the total number of tasks over those executors. Since each executor corresponds to a single thread, multiple tasks run serially on the same executor. However, each executor usually runs exactly one task (default option). Intra-worker and inter-worker communications are managed through queues. Each executor has its own input queue and output queue. Tuples are read from the input queue and processed by the thread handling the spout/bolt logic; they are emitted on the outgoing queue and then are moved to the parent worker's transfer queue by a send thread.

4 Constraint LTL over clocks with counters

The temporal logic model of Section 5 is expressed in terms of the CLTLoc logic [7] enriched with discrete unbounded counters, an extension of LTL allowing

arithmetical variables to occur in atomic formulae and be incremented or decremented by an integer value. The decision procedure for determining whether a CLTLoc formula with counters is satisfiable or not is at the basis of the prototype tool used in Section 6 to formally verify Storm topologies. In this section we define the logic and we provide a method to check the soundness of the outcome of the satisfiability procedure for the defined logic when a trace is returned. The assessment is partial, in the sense that if the produced trace does not pass the soundness check, then nothing can be said of the satisfiability of the formula until a model passing the check is found.

The logic allows for two kinds of atomic formulae. Atomic formulae over $(\mathbb{R}, \{<, =\})$ contain arithmetical variables which behave as clocks of Timed Automata [13]. For instance, a possible atomic formula over clock x is $x < 4$, where $x \in \mathbb{R}$. Atomic formulae over $(\mathbb{N}, \{<, =\}, +, 0, 1)$ predicate over arithmetical variables that have no semantic restrictions. For instance, an atomic formula of this second kind is $y + z < 4$, where both y and z are in $\mathbb{N}$.

A clock x measures the time elapsed since the last "reset" of x, which occurs when $x = 0$. Since the values of clocks can be compared with constants in constraints of the form $x \sim c$ (where $c \in \mathbb{N}$ and $\sim \in \{<, =\}$), clocks are used to constrain the time elapsing between relevant events of topologies. A counter y, instead, stores a value that can be incremented, decremented and tested against a constant value. We use counters to represent the size of bolts' queues. We also exploit the modality X applied to integer variables, introduced in [10]: if y is an integer variable, term $\mathrm{X}y$ represents the value of y in the next position of time.

Let V be a finite set of variables over $\mathbb{N}$. Atomic formulae θ over V are quantifier-free Presburger formulae over terms α of the form y or $\mathrm{X}y$, with $y \in V$.

Then, if C is a finite set of clock variables over $\mathbb{R}$, and AP is a finite set of atomic propositions, CLTLoc formulae with counters are defined as follows:

$$\phi := p \mid x \sim c \mid \theta \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}\phi \mid \mathbf{Y}\phi \mid \phi\mathbf{U}\phi \mid \phi\mathbf{S}\phi$$

where $p \in AP$, $x \in C$, $c \in \mathbb{N}$, $\sim \in \{<, =\}$, and $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{U}$ and $\mathbf{S}$ are the usual "next", "previous", "until" and "since" operators of LTL [13].

An *interpretation* of a formula is a pair (π, σ), where $\pi : \mathbb{N} \to \wp(AP)$, and $\sigma : \mathbb{N} \times \{C \cup V\} \to \mathbb{R}$ is a mapping associating every variable in $C \cup V$ with a value in $\mathbb{R}$, but restricting values of the elements in V to $\mathbb{N}$. The semantics of CLTLoc is defined as for LTL, except for formulae $x \sim c$ and θ. Let A_V be the ordered set of all terms of the form y and $\mathrm{X}y$, with $y \in V$, and let $n - 1$ be its cardinality; for each $\alpha_j \in A_V$, its depth $|\alpha_j|$ is such that $|\alpha_j| = 0$ if $\alpha_j = y$, and $|\alpha_j| = 1$ if $\alpha_i = \mathrm{X}y$ for some $y \in V$. Given a mapping $v : A_V \to \mathbb{N}$, $\theta[v(\alpha_0), \ldots, v(\alpha_{n-1})]$ is the valuation of θ through v, which is obtained by replacing each term α_j occurring in θ with value $v(\alpha_j)$. If $\theta[v(\alpha_0), \ldots, v(\alpha_{n-1})]$ is true we write $v \models \theta$. Let $t(\alpha_j) = y$ if α_j is either y or $\mathrm{X}y$. The following holds for each $i \in \mathbb{N}$, where the underlying assignment v is such that $v(\alpha_j) = \sigma(i + |\alpha_j|, t(\alpha_j))$:

$$\begin{aligned}
&(\pi, \sigma), i \models x \sim c \quad \textbf{iff } \sigma(i, x) \sim c \\
&(\pi, \sigma), i \models \theta \qquad \textbf{iff } \theta[\sigma(i + |\alpha_0|, t(\alpha_0)), \ldots, \sigma(i + |\alpha_{n-1}|, t(\alpha_{n-1}))]
\end{aligned}$$

If ϕ is a formula, interpretation (π, σ) is a *model* for ϕ if $(\pi, \sigma), 0 \models \phi$ holds.

The satisfiability problem for CLTL and CLTLoc is decidable [10,7] and can be practically computed through the Bounded Satisfiability Checking (BSC) technique [6,7]. In general, a BSC decision procedure, given a formula ϕ, looks for an ultimately periodic model of ϕ of the form $\alpha(s\beta)^\omega$, where $|\alpha s\beta| = k$. To achieve this, it looks to build a bounded structure of the form $\alpha s\beta s$, i.e., where a state s is repeated. In the case of LTL formulae, a state corresponds to a set of subformulae of ϕ. For CLTL (resp., CLTLoc) formulae, a state includes also arithmetic constraints capturing the relationships among variables (resp., clocks), even those that do not appear explicitly in the formula as atomic formulae. For these logics it is guaranteed that, when the decision procedure finds a structure of the form $\alpha s\beta s$ for formula ϕ, this can be extended to an infinite model of the form $\alpha(s\beta)^\omega$. These results, however, cannot be extended to CLTLoc augmented with counters, since the logic is in general undecidable, as it contains CLTL over quantifier-free Presburger formulae [11], i.e., the absence of ultimately periodic models for a formula does not entail its unsatisfiability.

As a consequence, we pursue a limited approach that stems from the analysis of the shape of the formulae defining the semantics of Section 5, which is still meaningful to discover possible dangerous executions of a Storm topology, i.e., those originated from a periodic behavior of its abstract model and representing undesired executions of running topologies (see Section 6). More precisely, we adapt the techniques developed in [6,7] into a procedure that, given a CLTLoc formula with counters and a bound k, tries to build a suitable structure $\alpha s\beta s$, with $|\alpha s\beta| = k$ and: (i) if no such structure is found, it concludes that no ultimately periodic models of length smaller than k exist; (ii) if a structure is found, it performs a check to determine whether the structure can be extended to an infinite model $\alpha(s\beta)^\omega$ and, if the check succeeds, it returns $\alpha s\beta$ as representative of the infinite model. If the check fails, the result is inconclusive, and a new structure must be looked for.

First of all, we remark that, since clocks and counters cannot be compared against each other, we can deal with them separately. In particular, the extendability *ad infinitum* of the assignments of values to clocks is guaranteed through the results of [7]. In the rest of this section, we outline a sufficient condition for extending *ad infinitum* a bounded assignment of values to counters.

In [10,6] the key abstraction that allows us to deal with the fact that variables have infinite domains is the notion of *symbolic valuation*, which captures the relationships between the values of the variables in a symbolic way. For example, if x, y, z are the variables appearing in formula ϕ, an example of symbolic valuation is the set of formulae $\{x < y, y < z, x < z\}$. In fact, symbolic valuations take into account also the fact that a CLTL formula can relate the values of variables at different time instants through the X operator. For example, if $x, \mathrm{X}y$ are the terms appearing in formula ϕ, an example of symbolic valuation is $\{x < y, x = \mathrm{X}x, \mathrm{X}x < y, \mathrm{X}x < \mathrm{X}y, y < \mathrm{X}y\}$. Notice that a symbolic valuation can contain formulae (and even terms) that do not appear explicitly in ϕ, such as $x = \mathrm{X}x$ in the previous example, in order to provide a complete picture of the

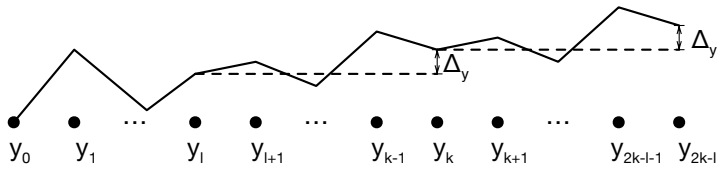

Figure 2: Example of repeated shape for the evolution of variable y.

relationships among variables over a sufficient horizon. Since in CLTLoc with counters we allow for richer constraints on variables (e.g., we can write formulae such as $\mathrm{X}x = 2x + y$), we cannot exhaustively capture the relationships among possible terms. Hence, we introduce the notion of *partial symbolic valuation* (p.s.v.). More precisely, given a formula ϕ such that Θ_ϕ is the set of all its atomic formulae over counters, its set of *partial symbolic valuations* pSV_ϕ is simply $\wp(\Theta_\phi)$. For example, if $\Theta_\phi = \{x < y, \mathrm{X}x = y + z, \mathrm{X}y < x + \mathrm{X}z\}$, an example of partial symbolic valuation is set $\{x < y, \mathrm{X}x = y + z\}$. Given a p.s.v. ρ_i, it *symbolically satisfies* an atomic formula θ iff $\theta \in \rho_i$, in which case we write $\rho_i \models_{psv} \theta$. We can extend the notion of symbolic satisfaction to sequences of p.s.v.'s and CLTLoc formulae with counters in a straightforward way; for example, if $\rho = \rho_0\rho_1 \ldots$ is a sequence of p.s.v.'s, $\rho, 0 \models_{psv} \mathbf{X}(\mathrm{X}x = y + z)$ iff $\rho_1 \models_{psv} \mathrm{X}x = y + z$. In addition, given a set A_V of terms, a mapping $v : A_V \to \mathbb{N}$, and a p.s.v. ρ_i, we say that v satisfies ρ_i, written $v \models \rho_i$ iff for each $\theta \in \rho_i$ it holds that $v \models \theta$. Notice that, given a mapping v and a set of formulae Θ_ϕ, v induces a maximal p.s.v., which is simply the set of all $\theta \in \Theta_\phi$ such that $v \models \theta$.

The goal of our decision procedure is, given a formula ϕ, to find a bounded sequence $\sigma_k : [0, k] \times V \to \mathbb{N}$ of assignments to variables—which in turn corresponds to a sequence of mappings $v_0v_1 \ldots v_{k-1}$ such that $v_i(y) = \sigma(i, y)$ and $v_i(\mathrm{X}y) = \sigma(i+1, y)$ for all $y, \mathrm{X}y \in A_V$—such that, if $\rho_0\rho_1 \ldots \rho_{k-1}$ is the sequence of maximal p.s.v.'s induced by $v_0v_1 \ldots v_{k-1}$: (i) there is $0 \leq l < k$ such that $\rho_0 \ldots \rho_{l-1}(\rho_l \ldots \rho_{k-1})^\omega, 0 \models_{psv} \phi$; (ii) σ_k can be extended to an infinite sequence of assignments $\sigma : \mathbb{N} \times V \to \mathbb{N}$, whose corresponding sequence of mappings $v_0v_1 \ldots$ is such that, for all $i \geq k$, it holds that $v_i \models \rho_{l+(i-k) \mod (k-l)}$.

This corresponds to finding a bounded sequence $\sigma_{k+1} : [0, k+1] \times V \to \mathbb{N}$, whose induced sequence of maximal p.s.v.'s $\rho_0\rho_1 \ldots \rho_k$ is such that $\rho_k = \rho_l$, and all subformulae of ϕ that hold at position l also hold at position k. In addition, as sufficient condition for the finite sequence of assignments to be extendable to an infinite one, we require that in the loop the evolution of each variable $y \in V$ has the same shape, as exemplified in Fig. 2. This entails that, for example, in the second iteration the value of y is the same as in the first iteration, plus the offset between the value of y in the first positions of the two iterations, represented as Δ_y in Fig. 2. Notice that, for the loop to be repeated *ad infinitum* with the same shape, Δ_y cannot be negative, since $y \in \mathbb{N}$.

For a bounded sequence σ_{k+1} to be extendable we check that, for each position i inside the loop (i.e., such that $l \leq i < k$), for each successive iteration n, with $n > 0$, for each $y \in V$, each atomic formula θ of ϕ has the same value whether y

is $\sigma_{k+1}(i,y)$ or $\sigma_{k+1}(i,y) + n\Delta_y$ (for example, if $\theta = y > 3$, $\sigma_{k+1}(i,y) = 5$, and $\Delta_y = 2$, both $\sigma_{k+1}(i,y) > 3$ and $\sigma_{k+1}(i,y) + n\Delta_y > 3$ hold). To perform the check, we ask whether Presburger formula (1) is satisfiable.

$$\forall n \left(\begin{array}{l} n > 0 \Rightarrow \\ \bigwedge\limits_{\substack{l \leq i < k \\ \theta \in \Theta_\phi}} \left(\begin{array}{l} \theta[\sigma_{k+1}(i,y_1), \sigma_{k+1}(i+1,y_1), \ldots, \sigma_{k+1}(i,y_m), \sigma_{k+1}(i+1,y_m)] \\ \Leftrightarrow \\ \theta[\sigma_{k+1}(i,y_1) + n\Delta_{y_1}, \ldots, \sigma_{k+1}(i+1,y_m) + n\Delta_{y_m}] \end{array} \right) \end{array} \right) \quad (1)$$

In Formula (1), the set of variables V is $\{y_1, \ldots, y_m\}$; the terms $\sigma_{k+1}(i, y_j)$ are constants defined by the sequence of assignments σ_{k+1} to check; $\theta[\sigma_{k+1}(i,y_1), \sigma_{k+1}(i+1,y_1), \ldots, \sigma_{k+1}(i,y_m), \sigma_{k+1}(i+1,y_m)]$ (resp. $\theta[\sigma_{k+1}(i,y_1) + n\Delta_{y_1}, \ldots, \sigma_{k+1}(i+1,y_m) + n\Delta_{y_m}]$) is the value of atomic formula θ when each term of the set A_V is replaced by its assigned value, where $A_V = \{y_1, \mathrm{X}y_1, \ldots, y_m, \mathrm{X}y_m\}$; and for each $y_j \in V$, $\Delta_{y_j} = \sigma_{k+1}(k, y_j) - \sigma_{k+1}(l, y_j)$. As mentioned above, if Formula (1) is false, then we cannot conclude that σ_{k+1} can be extended to an infinite model, nor that formula ϕ admits a model.

5 Formal Model of Storm Topologies

This section describes the CLTLoc (with counters)-based model of Storm topologies. We first outline the chosen abstraction level and assumptions and then we introduce the temporal logic model of each component. The model focuses on the behavior of the queues of the bolts of Storm topologies. It describes how the timing parameters of the topology, such as the delays with which tuples are input to the topology by spouts and the processing time of tuples for each bolt, affect the accumulation of tuples in the queues. We use clocks to capture timing features and counters to describe the evolution of the size of the queues.

Although the model refers to Storm topologies, for example in the assumptions made, it essentially consists of a set of nodes processing and exchanging information—more precisely, tuples—and storing incoming data in queues, formalized through counters. For this reason, we call this model an example of *timed counter network*, an abstraction for the behavior of Storm-like topologies.

The formal model allows for the definition of topologies in a compositional way, similarly to how topologies are created by code developers. We formalized the behavior of the relevant features and parameters of spouts and bolts by reverse-engineering the IRichSpout and IRichBolt interfaces and we used them as building blocks for creating topologies, under the following assumptions:

- Deployment details, such as the number of worker nodes and the features of the (possibly) underlying cluster are abstracted away; topologies are assumed to run on a single worker process and each executor runs a single task, which is the default configuration of the runtime, as described at the end of Section 3.
- Each bolt has a single receive queue for all its parallel instances and no sending queue, while the workers' queues are not represented, since we assume to be in a single-worker scenario. For generality, all queues have unbounded size.

(a) (b)

Figure 3: Finite state automata describing the states of spout (a) and bolt (b).

- We do not detail the contents of tuples, but only their quantities, since we measure the size of queues by the number of tuples they contain.
- The external sources of information from which spouts pull data are not explicitly represented, since they are outside of the perimeter of the application. Then, spouts are sources of tuples, so their queues are not represented.
- For each component, the duration of each operation or the permanence in a given state has a minimum and a maximum time.

A Storm topology is a directed graph $\mathbf{G} = \{\mathbf{N}, Sub\}$ where the set of nodes $\mathbf{N} = \mathbf{S} \bigcup \mathbf{B}$ includes in the sets of spouts ($\mathbf{S}$) and bolts ($\mathbf{B}$), and $Sub \subset \mathbf{B} \times \mathbf{N}$ captures the subscription relation defining how the nodes are connected to one another. If it holds that $(i, j) \in Sub$, this indicates that "bolt i subscribes to the streams emitted by spout/bolt j".

The behavior of both spouts and bolts can be illustrated by means of finite state automata (see Fig. 3). Spouts can be either emitting tuples or idle, therefore the corresponding automaton only has two states, *idle* and *emit*. Different emit actions (whose occurrence is captured by the system being in the *emit* state) can happen consecutively; also, the spout can be in the *idle* state for consecutive time instants. The possible execution sequences are determined by the timing constraints, as discussed in detail later. A bolt can alternatively be processing tuples, idle or in a failure state. The *process* macro-state is composed of three states, namely *take*, *execute* and *emit*. If a bolt is idle and its queue is not empty, it eventually reads tuples from the queue, performing an instantaneous *take* action, that is captured by the *take* state of the related finite state automaton. Immediately after a *take*, each bolt starts processing the tuples, an operation which lasts α time units, with α a parameter of the bolt, a positive real value which represents the amount of time that a bolt requires to process one tuple. This corresponds to the state *execute* in the automaton. Once the execution is completed, the bolt emits output tuples. This instantaneous action corresponds to the *emit* state. Bolts may fail and failures may occur at any moment; upon a bolt failure, the system goes to the *fail* state and all tuples stored, at that moment, in the queue of the failed bolt are lost, or replayed in case of a reliable topology. If no failure occurs, after an *emit* a bolt goes to *idle*, where it stays until it reads new tuples. Spout failures are not modeled; their effect is irrelevant for the growth analysis of bolt queues as they would reduce the workload on

the topology. Hence, our approach focuses only on the analysis of topologies processing a full workload, i.e., where spouts never fail.

We model the behavior of Storm topologies through a set of formulae of CLTLoc with counters. We refer to this logic-based model as *timed counter network*. We break this model down in four parts: (i) the evolution of the state of the nodes; (ii) the behavior of the counters (i.e., the queues); (iii) timing constraints; (iv) failures. We present here only some highlights of the model, whose full version can be found in [5].

State evolution. Each state is described through a combination of propositional variables. For example, a bolt j is in the macro-state *process* when $\texttt{process}_j$ holds. In addition, it is in *take* (resp. *emit*) state when $\texttt{take}_j$ (resp., $\texttt{emit}_j$) holds. The *execute* state, instead, corresponds to the configuration where $\texttt{process}_j$ is true while both $\texttt{take}_j$ and $\texttt{emit}_j$ are false. Formula (2) defines the conditions for $\texttt{process}_j$ to hold.

$$\bigwedge_{j \in \mathbf{B}} \left(\texttt{process}_j \Rightarrow \begin{pmatrix} \texttt{process}_j \, \mathbf{S} \, (\texttt{take}_j \vee (\texttt{orig} \wedge \texttt{process}_j)) \wedge \\ \texttt{process}_j \, \mathbf{U} \, (\texttt{emit}_j \vee \texttt{fail}_j) \ \wedge \ \neg \texttt{fail}_j \end{pmatrix} \right) \tag{2}$$

Queue behavior. We use $\mathbb{N}$-valued discrete counters to represent the amounts of tuples moving through the topology. Whenever a component is emitting tuples or reading from its queue, the related counters are updated according to several constraints. Every time $\texttt{emit}_j$ holds for a component j, $r_{\texttt{emit}_j}$ tuples are added to the queues of all bolts subscribing to j (i. e., the variables q_i representing the occupancy level of those queues are incremented by $r_{\texttt{emit}_j}$). When multiple components subscribed by a bolt emit tuples simultaneously, the increment on its queue is equal to the sum of all the tuples emitted, corresponding to the value of $r_{\texttt{add}_j}$. Dually, when $\texttt{take}_j$ holds, the occupancy level q_j is decremented by $r_{\texttt{process}_j}$ (number of tuples read by bolt j). Formulae (3)-(4) describe these situations. Notice that $\texttt{add}_j$ holds when at least one of the components subscribed by j is emitting, whereas $\texttt{startFail}_j$ is true in the first instant of a failure state.

$$\texttt{add}_j \wedge \neg \texttt{take}_j \wedge \neg \texttt{startFail}_j \Rightarrow (\mathrm{X} q_j = q_j + r_{\texttt{add}_j}) \tag{3}$$

$$\texttt{take}_j \Rightarrow (\mathrm{X} q_j = q_j + r_{\texttt{add}_j} - r_{\texttt{process}_j}) \tag{4}$$

The number of tuples extracted from the queue depends on the parallelism level of the bolt (i. e., the number of parallel executors as described in Section 3), that is represented in the model by the value of $\hat{r}_{\texttt{take}_j}$. When a *take* occurs, if the number of elements in the queue plus the ones being added in the current time instant is greater than $\hat{r}_{\texttt{take}_j}$, the variable representing the number of tuples that will be processed ($r_{\texttt{process}_j}$) is equal to $\hat{r}_{\texttt{take}_j}$, otherwise it is equal to $q_j + r_{\texttt{add}_j}$ (i. e., the bolt takes all elements from the queue). This captures how each bolt is able to concurrently process a number of tuples that is at most equal to the number of its executors.

$$(\texttt{take}_j \wedge \hat{r}_{\texttt{take}_j} \geq q_j + r_{\texttt{add}_j}) \Rightarrow (r_{\texttt{process}_j} = q_j + r_{\texttt{add}_j}) \tag{5}$$

$$(\texttt{take}_j \wedge \hat{r}_{\texttt{take}_j} < q_j + r_{\texttt{add}_j}) \Rightarrow (r_{\texttt{process}_j} = \hat{r}_{\texttt{take}_j}) \tag{6}$$

The number of tuples emitted by the bolt j ($r_{\texttt{emit}_j}$) at the end of the processing phase depends on parameter σ_j (a constant in $\mathbb{R}$), representing the ratio between output and input tuples. That is, given n_{in} input tuples, the total number of output tuples is equal to $\sigma \cdot n_{in}$. The value of σ is either measured by monitoring a deployed application, or defined by making assumptions based on the kind of operation performed by the bolt. Since $r_{\texttt{emit}_j} \in \mathbb{N}$, simply imposing $r_{\texttt{emit}_j} = \lfloor \sigma \cdot r_{\texttt{process}_j} \rfloor$ (resp., $r_{\texttt{emit}_j} = \lceil \sigma \cdot r_{\texttt{process}_j} \rceil$) may lead to excessive under- (resp., over-) approximation, especially when $0 \leq \sigma \cdot r_{\texttt{process}} \ll 1$. For this reason we keep track of the number of tuples processed, but not leading to the emission of output tuples. This is achieved through the auxiliary variable $\texttt{buffer}_j$, which is incremented as new tuples are correctly processed by the bolt. As formalized in Formula (7), when an *emit* occurs on bolt j, $r_{\texttt{emit}_j}$ is equal to $\lfloor \sigma \cdot \texttt{buffer}_j \rfloor$, and $\texttt{buffer}_j$ is then decremented by $\lfloor \frac{r_{\texttt{emit}_j}}{\sigma} \rfloor$. Conversely, Formula (8) defines that when the bolt is not emitting, `buffer` keeps its value until the next *emit*.

$$\bigwedge_{\substack{j \in B \\ \neg final(j)}} \left(\texttt{emit}_j \Rightarrow \begin{pmatrix} \texttt{buffer}_j = Y\texttt{buffer}_j + r_{\texttt{process}_j} & \wedge \\ r_{\texttt{emit}_j} \leq \sigma_j \texttt{buffer}_j & \wedge \\ r_{\texttt{emit}_j} > \sigma_j \texttt{buffer}_j - 1 & \wedge \\ X\texttt{buffer}_j \geq \texttt{buffer}_j - \frac{r_{\texttt{emit}_j}}{\sigma} & \wedge \\ X\texttt{buffer}_j < \texttt{buffer}_j - \frac{r_{\texttt{emit}_j}}{\sigma} + 1 & \end{pmatrix} \right) \tag{7}$$

$$\bigwedge_{j \in B, \neg final(j)} (\neg \texttt{emit}_j \Rightarrow (r_{\texttt{emit}_j} = 0 \wedge (X\texttt{buffer}_j = \texttt{buffer}_j) \mathbf{U} \mathbf{X} \texttt{emit}_j)) \tag{8}$$

Notice that some of the variables appearing in Formulae (3)-(8) have infinite domains, but some range over finite domains. More precisely, variables q_j for each bolt j, $r_{\texttt{add}_j}$ for bolts subscribing to spouts, and $r_{\texttt{emit}_i}$ for each spout i, are infinite counters. Variables $r_{\texttt{process}_j}$, instead, are finite counters since they have values between 0 and $\hat{r}_{\texttt{take}_j}$. Variables $\texttt{buffer}_j$ and $r_{\texttt{emit}_j}$ for each bolt, as well as $r_{\texttt{add}_j}$ for all bolts not subscribing to spout streams, are also finite counters. In fact, $\texttt{buffer}_j$, whose behavior is defined by Formulae (7) and (8), is finite since its value is always less than $\hat{r}_{\texttt{take}_j} + \frac{1}{\sigma} + 1$. We do not show the reasoning that allows us to conclude the finiteness of the aforementioned counters for lack of space. The finiteness of some of the counters allows us to write succinct formulae where multiplications and divisions are abbreviations for long case formulae.

Timing constraints. To measure the time spent in each state, and to impose timing constraints between different events, for each topology component we define a set of clock variables. Specifically, the duration of adjacent mutually exclusive processing phases (such as *idle*, *process* and *fail* for a bolt, *idle* and *emit* for a spout) is measured through two clocks, as done in [7]. At each instant only one of the two clocks is relevant to measure the time spent in the current processing phase; when the next phase starts, the second clock is reset and becomes the new relevant clock, while at the same time the value of the former is tested to verify if the measured delay satisfies the desired bound. In the following, we use a shorthand $t_{\texttt{phase}}$ to indicate the currently relevant clock. Formula (9) defines the conditions for resetting $t_{\texttt{phase}}$ for a bolt: in the origin, when a *take*

occurs, when a failure starts and when an *idle* phase starts.

$$t_{\texttt{phase}} = 0 \;\Leftrightarrow\; \texttt{orig} \vee \texttt{take} \vee (\texttt{fail} \wedge \neg \mathbf{Y}\texttt{fail}) \vee (\texttt{idle} \wedge \neg \mathbf{Y}\texttt{idle}) \quad (9)$$

Formula (10) imposes that when *emit* occurs, the duration of the current processing phase is between $\alpha - \epsilon$ and $\alpha + \epsilon$, where $\epsilon \ll \alpha$ is a positive constant that captures possible (small) variations in the duration of the processing.

$$\texttt{process} \wedge \texttt{emit} \Rightarrow (t_{\texttt{phase}} \geq \alpha - \epsilon) \wedge (t_{\texttt{phase}} \leq \alpha + \epsilon) \quad (10)$$

Measuring non-adjacent time intervals, such as the time between the end of a failure and the start of the next one (i. e., time to failure), can be done using a single clock, which does not need to be tested at the same time it is reset.

Failures. In our model, whenever a node fails, the tuples being processed by the node, together with the tuples in its receive queue, are considered as failed (not fully processed by the topology). According to the reliable implementation of Storm, the spout tuples that generated them must be resubmitted to the topology. Since we do not keep track of single tuples, but we only consider quantities of tuples throughout the topology, given an arbitrary amount of failed tuples we can estimate the amount of spout tuples that have to be re-emitted by the connected spouts. In order to express this relationship between the failing tuples in a specific (failing) node and the new tuples having to be re-emitted, we introduce the concept of *impact* of the node failure with respect to another (connected) node. $Imp(j,i)$ ("impact of node j failure on node i") is the coefficient expressing the ratio $\frac{tuples\_to\_be\_replayed(i)}{failed\_tuples(j)}$ where $j \in \mathbf{B}$ is the failing bolt and $i \in \{\mathbf{S} \bigcup \mathbf{B}\}$ is another node in the topology. If there exists a path $\{p_0, \ldots, p_n | n > 0, p_0 = i, p_n = j\}$ in the topology connecting the two nodes such that $\forall k \in [0, n-1] Sub(p_k, p_{k+1})$ holds, then a failure of node j has an impact on node i and $Imp(j,i) > 0$. If such a path does not exist, then $Imp(j,i) = 0$. The procedure to obtain the values of $Imp(j,i)$ for each bolt is described in [5]. Once this coefficient is calculated for all pairs of $(bolt, spout)$ in the topology, it allows us to determine $r_{\texttt{replay}_i}$, (i. e., the number of tuples to be re-emitted by spout i after a bolt failure) by simply multiplying the number of failed tuples by the appropriate coefficient, as $\bigwedge_{i \in \mathbf{S}} (r_{\texttt{replay}_i} = \sum_{j \in \mathbf{B}} r_{\texttt{fail}_{ji}} \cdot Imp(j,i))$, where $r_{\texttt{fail}_{ji}}$ expresses "the number of failed tuples in bolt j affecting spout i". This value is incremented as in Formula (11) whenever a failure starts and is reset after all the $r_{\texttt{fail}_{ji}} \cdot Imp(j,i)$ tuples are emitted by the spout. Interested readers can refer to [5] for the complete model.

$$\bigwedge_{i \in \mathbf{S}, j \in \mathbf{B}} (\texttt{startFail}_j \wedge \neg \texttt{emit}_i \Rightarrow \mathrm{X} r_{\texttt{fail}_{ji}} = r_{\texttt{fail}_{ji}} + q_j + r_{\texttt{process}_j} + r_{\texttt{add}_j}) \quad (11)$$

6 Experimental results

We present some experimental results obtained with our prototype tool **D-VerT**[3], whose architecture is described in [5]. As shown in Fig. 4, **D-VerT** takes as

[3] github.com/dice-project/DICE-Verification

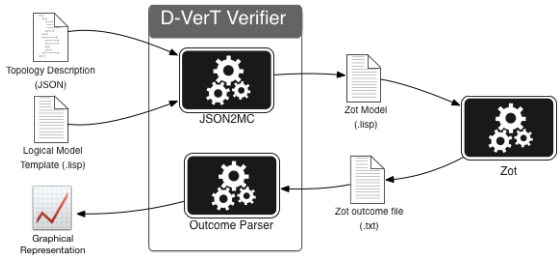

Figure 4: **D-VerT** verification flow.

input the description of a Storm topology, through a suitable JSON format, and implements the model-to-model transformation which produces the corresponding instance of timed counter network representing the topology. The resulting model is fed to the Zot formal verification tool [2], which has been modified to deal with CLTLoc formulae including unbounded counters. The property is violated if a non-spurious counterexample (i.e. a run of the system violating the property) is found. In this case, Zot returns the violating trace (*SAT* result), that is processed back and displayed graphically by **D-VerT**. If the verification terminates without providing counterexamples (*UNSAT* result), then the property holds limited to ultimately periodic executions represented by a prefix $\alpha s \beta s$ of bounded length.

We consider two different topologies: a simple DIA and a more complex topology (named "focused-crawler") provided by an industrial partner within the DICE consortium. In both cases, we verify the property "*all bolt queues have a bounded occupation level*". If the property holds, then we claim that all bolts are able to process the incoming tuples in a timely manner. Otherwise, there exists a counterexample that violates (i.e., disproves) the property and that corresponds to an unwanted execution of the topology where at least one queue grows with an unbounded trend. This behavior can be expressed in the k-satisfiability problem with a formula constraining the size of the queues. Over ultimately periodic executions, defined through a k-bounded model, a queue q grows indefinitely if its size at position k is strictly greater than the size at position l. Therefore, to enforce the construction of models satisfying such a constraint, we add to the formulae defining the k-satisfiability the conjunct $\bigvee_{j \in \mathbf{B}} q_j(l) + c < q_j(k)$, where c is a non-negative constant.

The first use case (depicted in Fig. 1) allowed us to test some basic structures that may appear in a Storm topology, such as split and join of multiple streams. On this topology, we experimented on how modifying the parallelism level of a bolt affects its ability of processing incoming tuples. In the first analysis, run with the configuration in Fig. 1, Zot produces a trace showing that the adopted configuration leads to an unbounded increase of the queue occupation of B_2 and B_3. By changing the parallelism level of the bolts (setting it to, respectively, 8 for B_2 and 5 for B_3) we obtain a configuration showing no counterexample (up to length $k = 15$) of unbounded queue increase (timings of the two configurations – *simple-DIA-cfg-1* and *simple-DIA-cfg-2* – are reported in Table 1).

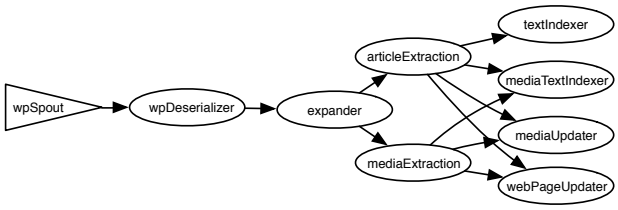

Figure 5: "focused-crawler" topology.

Topology	Bolts	Time	Max Memory	Outcome	Spurious
simple-DIA-cfg-1	3	60s	104MB	SAT	no
simple-DIA-cfg-2	3	1058s	150MB	UNSAT	N/A
focused-crawler-complete	8	2664s	448MB	SAT	no
focused-crawler-reduced-cfg-1	4	95s	142MB	SAT	no
focused-crawler-reduced-cfg-2	4	253s	195MB	SAT	no
focused-crawler-reduced-cfg-3	4	327s	215MB	SAT	no
focused-crawler-reduced-cfg-4	4	333s	206MB	SAT	no
focused-crawler-reduced-cfg-5	4	3184s	317MB	SAT	yes
focused-crawler-reduced-cfg-6	4	1060s	229MB	SAT	yes

Table 1: Experimental analysis on commodity hardware (MacBook Air running MacOSX 10.11.4. with Intel i7 1.7 GHz, 8 GB 1600 MHz DDR3 RAM; SMT solver used by Zot was z3 v.4.4.1). The complete results and experimental configurations can be found at dice-project.github.io/DICE-Verification.

The second use case represents a typical usage of Storm in big data applications. As part of a social network analysis framework, the topology depicted in Fig. 5 is in charge of fetching and indexing articles and multimedia items from multiple web sources. The formal analysis of the "focused-crawler" topology is motivated by some concerns raised by the industrial partner that were witnessed by monitoring the deployed application. After running the verification on the topology we pointed out the critical role of the *expander* bolt. Some output traces show possible system executions, even without failures, where the queue occupation level of such component is unbounded. Fig. 6 shows two of the graphical output traces provided by **D-VerT** (referring to bolts *expander* and *wpDeserializer*). It can be noticed, by looking at the number of tuples in the queues (black solid lines) over time, how they both represent a periodic model in which a suffix (in gray) of a finite sequence of events is repeated infinitely many times after a prefix. After ensuring that the trace is not a spurious model, we concluded that the expander queue, having an increasing trend in the suffix, is unbounded. In order to evaluate the performance and the scalability of the tool, we carried out many experiments on the presented topologies, by varying the topology parameters and the number of bolts considered. Table 1 shows some of the time and memory consumptions statistics we collected.

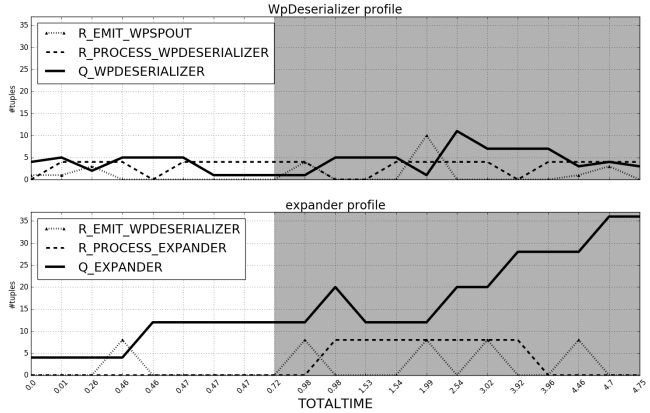

Figure 6: **D-VerT** output trace of bolts *expander* and *wpDeserializer*. Black solid lines represent the number of tuples in each bolt queue over time. Dashed lines show the processing activity of the bolt, and dotted lines show the *emit*s from the component upstream. Gray background highlights the suffix of the trace, that is repeated infinitely many times.

It can be noticed how the running time is strongly affected by both the number of bolts and their configurations, while the memory consumption is mainly correlated to the topology size (therefore, the number of formulae in the model). In the simple-DIA case study, we obtained counterexamples (*SAT* results) with very different timings depending on the configuration. The configuration leading to the *UNSAT* result, discussed previously, took considerably more time to terminate. In the "focused-crawler" case study, we ran the verification also on subsets of the topology (*focused-crawler-reduced*). In some cases, the tool provided a spurious counterexample. Despite the long running times in some cases, we think that the experiments show the feasibility of our approach, and we will focus in the future to optimizing the efficiency of the tool.

7 Conclusions and Future Work

In this paper we proposed a tool-supported approach for the formalization and automated verification of DIAs based on Storm technology. We presented a formal model of the temporal behavior of Storm topologies expressed through formulae of the CLTLoc extended with counters. We implemented a prototype tool, **D-VerT**, which takes as input a high-level description of the target topology, produces the corresponding set of logic formulae, and carries out the verification task via the Zot bounded satisfiability checker. We evaluated the tool through a pair of case studies.The running times of the tool range from a few minutes to hours, depending on the topology and on the configuration parameters. Since

the satisfiability of CLTLoc with counters is generally undecidable and the tool introduces some approximations to make the verification feasible in practice, we provided a procedure to determine, given a trace returned by the tool, whether this is spurious or not.

Future extensions and improvements of this work will follow several directions. In particular, we plan to: *(i)* extend the range of properties to be analyzed for the target topologies; *(ii)* pursue a finer-grained modeling approach, for example representing the internal messaging system with higher detail, to support more precise analyses; *(iii)* model other relevant technologies, such as Apache Spark and Apache Tez, by extending the current framework; *(iv)* further study the current model from a theoretical point of view, to achieve new results on the soundness and completeness of the analysis of timed counter networks.

Acknowledgment Work supported by Horizon 2020 project no. 644869 (DICE).

References

1. Apache Storm, `http://storm.apache.org/`
2. The Zot bounded satisfiability checker. github.com/fm-polimi/zot
3. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. In: Proceedings of LICS. pp. 160–170 (1993)
4. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of the expressiveness of timed automata and time petri nets. In: Proceedings of FORMATS. pp. 211–225 (2005)
5. Bersani, M., Erascu, M., Marconi, F., Rossi, M.: DICE verification tool - initial version. Tech. rep., DICE Consortium (2016), www.dice-h2020.eu
6. Bersani, M.M., Frigeri, A., Morzenti, A., Pradella, M., Rossi, M., Pietro, P.S.: Constraint LTL satisfiability checking without automata. J. Applied Logic 12(4), 522–557 (2014)
7. Bersani, M.M., Rossi, M., San Pietro, P.: A tool for deciding the satisfiability of continuous-time metric temporal logic. Acta Informatica 53(2), 171–206 (2016)
8. Bouajjani, A., Mayr, R.: Model checking lossy vector addition systems. In: Proceedings of STACS. LNCS, vol. 1563, pp. 323–333 (1999)
9. Casale, G., Ardagna, D., Artac, M., Barbier, F., Nitto, E.D., Henry, A., Iuhasz, G., Joubert, C., Merseguer, J., Munteanu, V.I., Perez, J., Petcu, D., Rossi, M., Sheridan, C., Spais, I., Vladušič, D.: DICE: Quality-driven development of data-intensive cloud applications. In: Proc. of MiSE. pp. 78–83 (2015)
10. Demri, S., D'Souza, D.: An automata-theoretic approach to constraint LTL. Information and Computation 205(3), 380–415 (2007)
11. Demri, S., Gascon, R.: The effects of bounding syntactic resources on Presburger LTL. Tech. Rep. LSV-06-5, LSV (2006)
12. Finkel, A.: Decidability of the termination problem for completely specified protocols. Distributed Computing 7(3), 129–135 (1994)
13. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling Time in Computing. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2012)
14. Karp, R.M., Miller, R.E.: Parallel program schemata. Journal of Computer and System Sciences 3(2), 147 – 195 (1969)
15. Reutenauer, C.: The Mathematics of Petri nets. Masson and Prentice (1990)