Formally Verifying the Absence of Overloading in Apache Storm*

Elena Pagani[1,†], Marcello M. Bersani[2], Mădălina Eraşcu[3], Matteo G. Rossi[4] and Silvio Ghilardi[5]

[1]Università degli Studi di Milano, via G. Celoria 18, Milano, 20133, Italy
[2]Politecnico di Milano, Via G. Ponzio 34, Milano, 20133, Italy
[3] West University of Timişoara, 4 Vasile Pârvan Blvd., 300223 Timişoara, Romania
[4]Politecnico di Milano, Via Privata Giuseppe La Masa 1, Milano, 20156, Italy
[5]Università degli Studi di Milano, via C. Saldini 50, Milano, 20133, Italy

Abstract

Apache Storm is widely used for real-time data stream processing in critical domains such as healthcare, logistics, monitoring and enforcement of Internet security, where maintaining performance under high or adversarial workloads is essential. However, modelling it so as to resort to formal verification techniques has proven to be a daunting task.

This paper addresses the formal verification of Storm configurations against overloading, expressed as bounded queue growth under given timing and parallelism parameters. We introduce a novel finite-state modelling strategy that preserves essential Storm behaviour, including nondeterministic outputs and timed processing, while abstracting indistinguishable tuples and threads into counters and advancing time by sound leaps between relevant events. The resulting models are generated for NuSMV, hence checked with unbounded symbolic model checking. Experiments on topologies of increasing complexity show that safe configurations are verified without memory exhaustion, and that counterexamples expose unsafe parameter choices.

Keywords

Real-time data analysis, Storm topology, Symbolic Model Checking, Robustness to overloading

1. Introduction

Storm is a framework for distributed real-time big data analytics. It was created by Nathan Marz who released the first version of the software in September 2011. In September 2014, Storm became an Apache Top-Level Project; in April 2026, version 2.8.6 was released. Apache Storm [3] (simply *Storm* hereafter) is a free and open-source project that is currently used by a number of companies [4] – among which Yahoo!, Twitter, Spotify and Groupon – that have integrated it into their IT infrastructures. Storm can be used with any programming language; it supplies APIs to extract data from databases and manipulate them. Storm can supply very high throughput by means of inherent parallelism in processing data items; it is fault-tolerant thanks to the ability of restarting processing in the event of an interruption and possibly moving it to a different machine in case of a failure, and it aims at guaranteeing an **at-least-once processing semantics**, that is, no data item is ever lost. Amongst its

CILC 2026: The 41st Italian Conference on Computational Logic, June 23–25, 2026, Ferrara, Italy

*This work was supported in part by the project *EcoCyber* included in the *Spoke 08 - Risk Management & Governance* of the Research and Innovation Program PE00000014, *"SEcurity and RIghts in the CyberSpace (SERICS)"*, under the National Recovery and Resilience Plan funded by the European Union - NextGenerationEU; and in part by PSR2024 funded by Università degli Studi di Milano. The fifth author is supported by INdAM's group GNSAGA.

† Corresponding author.
elena.pagani@unimi.it (E. Pagani); marcellomaria.bersani@polimi.it (M. M. Bersani); madalina.erascu@e-uvt.ro (M. Eraşcu); matteo.rossi@polimi.it (M. G. Rossi); silvio.ghilardi@unimi.it (S. Ghilardi)
https://homes.di.unimi.it/~pagae/ (E. Pagani); https://bersani.faculty.polimi.it/ (M. M. Bersani); https://merascu.github.io/ (M. Eraşcu); https://www.mecc.polimi.it/en/staff/matteo.rossi (M. G. Rossi); http://users.mat.unimi.it/users/ghilardi/ (S. Ghilardi)
0000-0001-7162-5997 (E. Pagani); 0000-0001-5137-940X (M. M. Bersani); 0000-0002-0435-5883 (M. Eraşcu); 0000-0002-9193-9560 (M. G. Rossi); 0000-0001-6449-6883 (S. Ghilardi)

use cases there are **real-time processing** of unbounded streams of data, online machine learning, execution of distributed Remote Procedure Calls, and "Extract, Transform, and Load" procedures for data integration from multiple sources. Provided these characteristics, Storm is a viable solution also for critical infrastructures which require real-time processing of huge amount of data to obtain prompt recommendations on how to operate. For instance [4]: Storm is used by The Weather Channel to collect data and produce weather forecasts. Verisign uses Storm to monitor the functioning of the Internet, to ensure security, stability, and resilience of key Internet infrastructures and services, and to mitigate cyber attacks. Cerner uses Storm to process massive amounts of clinical data in real-time. Aeris leverages Storm to support IoT applications involving machine-to-machine communications, where machines may be vehicles or cloud-based swarms of robots.

In critical infrastructures, it is often vital not to lose data, to fulfil strict real-time constraints on the achievement of data processing results, and to rely on a platform whose behaviour is *stable*, also when dealing with peaks of submitted workload [11]. The nominal properties of Storm applications rely on the correct deployment and configuration of the infrastructure, ensuring their validity even in the face of exceptionally adverse conditions.

This work considers the problem of validating the safety of a Storm configuration in terms of its robustness to *overloading*, which is defined as the ability of coping with certain workload levels avoiding that the queues containing the data to be processed become saturated or grow continuously. Overloading occurs when the data arrival rate is higher than the data processing rate, and it may lead to either the increase of the processing time resulting in violation of real-time constraints, or data loss due to memory overflow. From a practical point of view: the length of a message queue is determined by the value of the variable `topology.executor.receive.buffer.size` [5], which may be set by the topology implementers based on estimates/forecasts of the maximum accepted incoming traffic. The usefulness of solving this problem lies in *performing a proper a-priori tuning of parameters* for a certain topology so as to guarantee (i) compliance to service requirements; and (ii) a certain level of robustness[1] of a Storm infrastructure against potential Denial-of-Service attacks designed to compromise its accuracy through the submission of artificially elevated loads. The analysis is conducted using formal verification methods, modelling a Storm configuration as a finite-state system. A negative result of the verification provides a sort of recommendation on either the use of a larger buffer size or the adoption of additional mechanisms such as the *Backpressure Wait strategy* [5].

The problem was already investigated in past works (see Section 2), using simplified representations of the Storm nodes behaviour, or using formalisms subject to produce spurious traces that must then be checked with a further step, or using formalisms that have proven to be incapable to converge.

In this work, we propose a *novel strategy to model Storm applications* that, although *accurate*, makes the verification affordable and concludes the computation in feasible times. The contributions of this work are:

- the analysis of the problems involved in modelling Storm applications and studying its properties with formal methods;
- the proposal of a novel strategy to model a Storm application as a finite-state system whose verification with (*unbounded*) model checking tools scales well and is computable in reasonable time;
- the validation of the proposed strategy through its use to model several Storm applications of increasing complexity, and the measurement of verification performance in terms of both latency in obtaining the outcome and memory overhead of the system.

The work is structured as follows: in Section 2, an overview of the related work in the literature is supplied. Section 3 describes the characteristics of the Storm infrastructures and discusses the considered problem. Section 4 introduces the formalism, the notation used throughout the work, and the tool adopted for the verification of the considered systems. Section 5 describes the proposed modelling technique and explains the models' structure in detail. Section 6 discusses the results obtained by

[1] I.e., ability to continue functioning correctly even in the face of unexpected or abnormal situations.

running the produced models, in terms of both verification and performance results. Section 7 concludes the work.

2. Related Work

There are a few articles specifically considering the formal verification of Apache Storm. In [13], the authors face the modelling of Storm configurations and their verification. Storm is modelled using the CLTLoc metric temporal logic extended with counters; safety is then checked with the Zot bounded satisfiability checker [21] using Z3 [8] as SMT solver. Yet, processing nodes (*bolts*) are modelled such that the threads simultaneously take queued tuples, and simultaneously emit them at the end of the processing, with no further take possible on behalf of an idle thread when others are executing; this may lead to unemployed threads on one hand, and to tuples unnecessarily piling up in the queues, waiting for the next take, on the other hand. Furthermore, in order to overcome the fact that the satisfiability of CLTLoc with counters is generally undecidable, the tool introduces approximations that may produce spurious traces, and an additional procedure must thus be applied to validate the traces afterwards. By contrast, our strategy accurately represents the parallelism within a bolt not forcing all of its threads to proceed in lockstep; it considers traces whose length is not bounded a-priori,[2] and it is immune to the problem of spurious traces.

In [14], the verification engine used in [13] was encapsulated into a model-driven engineering tool (D-VerT) that can produce Storm models from UML diagrams and supply the Zot output in either textual or graphical format. In [15], the authors work further on the topic by showing how D-VerT can be used in a design–verify loop leading to a safe configuration. Though, the underlying logical framework is unchanged.

In [6], the authors investigated the application of the array-based formalism to model and validate Storm configurations, considered as infinite-state systems: the aim was to prove the safety of a configuration *for any number of threads in bolts*. However, various problems arose inside such infinite state specification and the tools employed (MCMT [9] and Cubicle [7]) were unable to converge due to memory exhaustion. Successive unpublished work (refining the previous formalisation) revealed also an even more structural problem: as predicted in certain literature [1], the presence of universal guards in transitions may produce spurious traces, a phenomenon that may occur in systems which are safe but whose safety cannot be certified by invariants expressible in universal first order logic [10].

Given the serious consequences that security breaches or malfunctions of critical ICT infrastructures may have, the use of formal verification techniques and model checking to assess properties of real software and systems is spreading. These techniques have been applied in broader fields including cloud infrastructures [19], security protocols [22], Internet protocols [18], Software Defined Networking [20], multi-robot systems [12].

3. Storm Topology and Problem Specification

In Storm [3], computation is performed by a *topology* (Figure $1(a)$), which indicates how data *streams* flow through nodes; a stream is an unbounded sequence of data items (*tuples*). Nodes are of two types: *spouts* and *bolts*. Spouts inject streams into the topology; they can get the data from external sources. Tuples are consumed by bolts, which process them so as to produce either partial results to pass on to other bolts, or final results that exit the system. Nondeterminism may show up in spouts' behaviour: in Figure $1(a)$, S_1 may emit one of its tuples to either B_1 or B_2 (but not both). In the following, the term *fan-in* is used to indicate the number of edges entering a bolt.

Parallelism is implemented and adjusted inside bolts by spawning a certain number of *threads* (Figure $1(b)$). We use the same assumptions as in [6], i.e., each bolt has one queue where tuples are enqueued, and from which threads extract tuples to be processed. Each thread can process one tuple at a time.

[2] In [13], a bound $k = 15$ was used. In our experiments, the shortest unsafe trace was 34 states long, observed with the *convIn* topology.

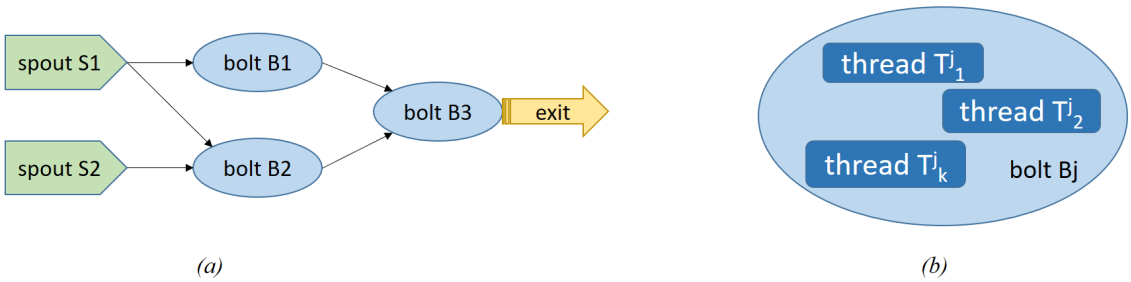


Figure 1: (a) Example of Storm topology. (b) Inner structure of a bolt.

A thread that is processing a tuple does not extract another one from the queue until it has finished processing the current one. A thread that has no tuple to process (i.e., whose bolt queue is empty) is *idle*.

More formally: for a certain topology $\mathcal{T}$, the symbols $\mathcal{S}$ and $\mathcal{B}$ indicate the number of spouts and the number of bolts respectively. S_i, $i \in [1, \mathcal{S}]$, denotes the i-th spout; B_j, $j \in [1, \mathcal{B}]$, the j-th bolt; T_k^j the k-th thread in bolt B_j. The parameters determining the behaviour of the nodes in $\mathcal{T}$ are: the interval of time between two consecutive emissions of a tuple performed by a spout S_i to one of the bolts connected to it (indicated with ES_i); the time spent by a thread in B_j to process a tuple (EB_j), the number m_j of threads in B_j.

3.1. Problem Specification

The **safety problem** associated with a Storm topology consists in *guaranteeing that the length of the bolts queues never grows beyond a certain threshold under a certain set of parameters.* This property implies that tuples do not risk to starve, that is, they are processed within finite time, which qualitatively implies real-time guarantees.

A Storm topology belongs to the class of *reactive parameterized systems*; in fact, it is parameterized in the number of threads, and it is reactive as threads behaviour is dictated by the emission of tuples by spouts and other threads. In this work, we treat a Storm topology as a *finite-state system* due to the finite number of spouts, bolts and threads, and the (imposed by design) boundedness of the bolts' queues; when bounds are exceeded, the system enters into a final error state. Each spout has a clock (Timed Automata style [2]) that measures the time between the emission of a tuple and the next. Similarly, the time that passes between the start and end of a thread's processing of a tuple is measured. For each spout S_i, its clock is reset cyclically so that its value is always in $[0, ES_i]$, and the same holds for the clocks of the processing threads. Communication between elements of the topology has negligible latency compared to the computation time of bolts and spouts. Therefore, all arithmetic calculations occur within finite intervals known a-priori.

The safety problem can be formally written as follows: if Q_j indicates the desired upper bound on the length of B_j's queue, then the safety condition is: $\bigwedge_{j \in [1,\mathcal{B}]} queue(B_j).length \leq Q_j$ where, for each B_j, Q_j should not be lower than B_j's fan-in in order to avoid trivial unsafety due to simultaneous emissions of tuples by all the upstream nodes.

It is worth to notice that increasing the computational power by increasing the number of threads in bolts is *not* a trivial solution to cope with the above problems. Indeed, it may have an unwanted side effect that in this work is indicated as *funnel effect*, resulting in transforming a safe configuration into an unsafe one. Let us consider the Storm topology in Figure 2, and let us suppose that ES_i is much greater than any EB_j, for every spout S_i; for instance: $\forall i \in [1, 3]$, $ES_i = 10$ clock ticks, while $\forall j \in [1, 2]$, $EB_j = 1$ and $m_j = 1$. For the considerations above, the safety property might be $queue(B_1).length \leq 3 \wedge queue(B_2).length \leq 1$. If all spouts emit simultaneously at $t = 0$ then $queue(B_1).length = 3$. At $t = 1$, B_1 emits a tuple towards B_2 and T_1^1 starts processing the second tuple; at $t = 2$, B_2 emits its first tuple while B_1 emits the second tuple towards B_2 – which is taken in

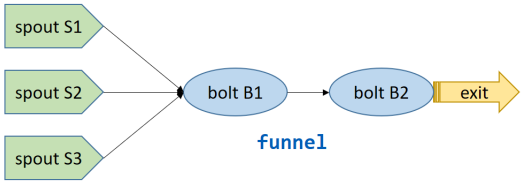


Figure 2: Example of Storm topology vulnerable to the funnel effect.

charge by T_1^2 – while T_1^1 starts processing the third tuple; at $t = 3$, B_1 emits the third tuple and its thread becomes idle, while B_2 emits the second tuple and starts processing the third one; at $t = 4$, B_2 emits the third tuple and its thread becomes idle. This configuration is *safe*.

If we increase the computational power in B_1 setting $m_1 = 3$ threads, then: if all spouts emit simultaneously at $t = 0$ then $queue(B_1).length = 3$ and all threads in B_1 start working; at $t = 1$ all the three tuples are emitted towards B_2, whose queue length increases beyond the upper bound, thus leading the system in an *unsafe* state.

4. Finite State Verification

NuSMV [16] is a symbolic model checker combining Binary Decision Diagrams (BDDs) and SAT-based model checking. Our system specification is finite state but nevertheless requires performing some arithmetic inside bounded intervals. For this reason, we adopted NuSMV that is a well-established, robust symbolic model checker which appears particularly suitable for our problem.

In the following, the primed notation x' is used to indicate the new value of a variable x; the assignment of a new value to a variable is represented by the symbol $\leftarrow$. This notation corresponds to the `next` operator in NuSMV syntax.

A system is specified via a pair of formulæ $\iota(\underline{p})$ and $\tau(\underline{p}, \underline{p}')$, where $\underline{p}$ is the set of variables determining the state of the system, $\iota(\underline{p})$ is the formalization of possible initial states of the system, $\tau(\underline{p}, \underline{p}') := \bigvee_{i=1}^{n} \tau_i(\underline{p}, \underline{p}')$ symbolizes the n possible state transitions of the system that modify $\underline{p}$ into $\underline{p}'$. A safety problem is given by a further formula $\upsilon(\underline{p})$, which describes the set *Bad* of states verifying the unsafe condition. Each transition τ_i is composed by a *guard* and a set of updates: if the current values of the variables satisfy the guard, then the transition can fire and the updates are applied. More guards may be verified simultaneously. In this case, one of the corresponding transitions fires nondeterministically. A *safety model checking problem* is the problem of checking whether the formula

$$(\star)_w \qquad \iota(\underline{p}_0) \wedge \tau(\underline{p}_0, \underline{p}_1) \wedge \cdots \wedge \tau(\underline{p}_w, \underline{p}_{w+1}) \wedge \upsilon(\underline{p}_{w+1})$$

is satisfiable for some $w \in \mathbb{N}$, $w \geq 1$, that is, whether a state in *Bad* can be reached in w steps from an initial state by applying the possible transitions.

5. Modelling a Storm Topology

This work proposes a novel modelling strategy that enables the verification of a Storm topology without losing accuracy of the models. The strategy consists in *(i)* abstracting some irrelevant details of the threads behaviour, and in *(ii)* speeding up the elapsing of the time whenever possible.

The abstraction is based on two considerations about properties intrinsic to real Storm applications. First, **tuples are indistinguishable from each other, and so are threads**. This is because every tuple undergoes processing in an equal amount of time, irrespective of the specific thread within a bolt responsible for processing it. Moreover, all threads behave in the same manner when processing tuples. Therefore, identifying individual tuples and threads is irrelevant: the crucial aspect is keeping track of the number of tuples queued and of the number of threads in a certain state: *idle*, *processing*, *ready to*

emit. Second, there are **periods of downtime during which nothing significant happens**, where relevant events are: emission of a tuple by either a spout or a bolt; threads becoming idle and thus able to take in charge a new tuple.

According to the former consideration, instead of modelling the completion of each individual thread's computation within a bolt with a real value in $[0, 1]$ (as in [6]), the new modelling (i) distinguishes only a finite number of different completion levels, and (ii) keeps track of only the number of threads that completed the same amount of computation, with a finite number of discrete counters. In other words, for each bolt B_j, threads are counted as belonging to one of $b + 2$ bins (with $b > 1$ selected by the modeller): b bins include the threads that completed the same percentage of computation, and correspond to discrete counters. Counters Td_j, with d representing the percentage of performed processing and assuming values p/b for $p \in \mathbb{N}, 1 \leq p \leq b - 1$, count the threads in bolt B_j that have completed $d\%$ of the processing of the assigned tuple; counter $T100_j$ counts the threads that have completed the computation. The counters for the last two bins are TI_j for idle threads, and $T0_j$ for threads that have just started the computation.

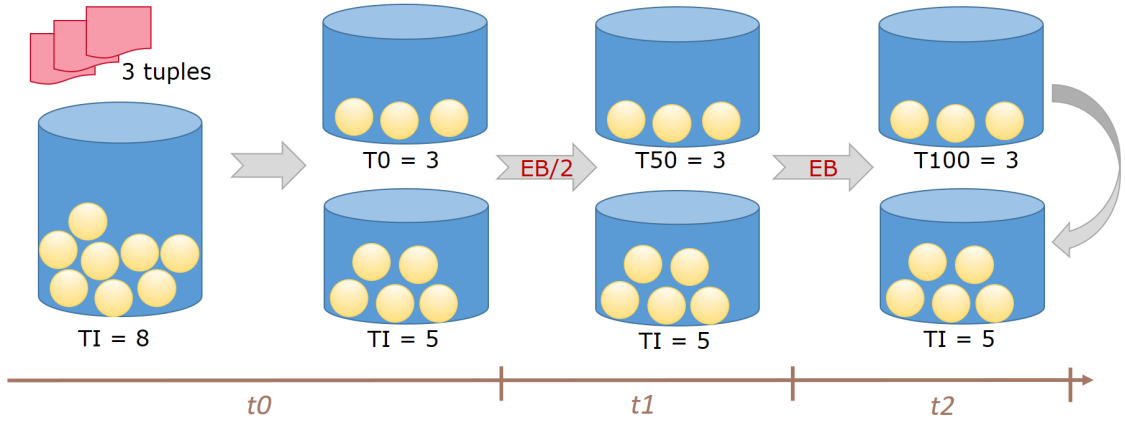


Figure 3: Abstraction of the tuple processing mechanism inside one bolt.

Figure 3 depicts the implemented abstraction. Let us assume that $b = 2$, $EB = 2$ for all threads of a bolt B, and that B receives tuples from one spout S. Let us assume that at a time $t = 0$ there are 3 tuples queued in B, and there are $TI = 8$ idle threads: $T0 = 3$ threads may start processing all the queued tuples, and the number of idle threads becomes $TI = 5$. When $t = 1$, the working threads are at 50% of their processing time; at time $t = 2$ they have finished processing and are ready to emit the processed tuples and return to idle state. The fourth counter besides TI, $T0$, $T100$, is $T50$, counting the threads that are halfway in their tuple processing. With each tick of the clock, the counters are updated as follows: $T100' \leftarrow T50 \wedge T50' \leftarrow T0 \wedge T0' \leftarrow 0 \wedge TI' \leftarrow TI + T100$ thus simulating the movements of threads from one bin to another. Furthermore, with each tick of the clock a timer TS is incremented that counts the time until the next S emission. When $TS = ES$, the B queue may be incremented of 1, and $TS' \leftarrow 0$. Idle threads taking in charge queued tuples are modelled as counters updates as well, according to the following pseudo-code:

```
if (queue(B).length > 0 ∧ TI ≥ queue(B).length) then
    TI' ← TI − queue(B).length;
    T0' ← queue(B).length;
    queue(B).length' ← 0;
if (queue(B).length > 0 ∧ TI < queue(B).length) then
    TI' ← 0;
    T0' ← TI;
    queue(B).length' ← queue(B).length − TI;
```

The first `if` block considers the case of more idle threads than queued tuples: the queue is emptied, and as many idle threads as many tuples move to processing state. Otherwise, the queue length is decreased of the number of idle threads, and all idle threads move to processing state.

5.1. Time Granularity in Case of Generic EB

According to the latter consideration above and depending on both ES and EB, it may happen that relevant events do not occur at every clock tick. Let us consider a topology with one spout having $ES = 9$ and one bolt having $EB = 6$, and let $b = 2$ hold. With such a modelling setting, a processing thread is moved from $T0$ to $T50$ and from $T50$ to $T100$ *every 3 clock ticks*, because two are the shifts that threads undertake. Analogously, counter TS can soundly be incremented every 3 clock ticks and – every 3 increments – it reaches ES, a tuple emission occurs and TS is set to 0. This mechanism, consisting in modelling the progression of time in discrete **leaps**, is *accurate* in that all relevant events occur at the exact time, that is, no event is anticipated, postponed, or disregarded. By contrast, building a model where time is incremented with the granularity of 1 clock tick does not supply greater accuracy, but involves greater computation time for the verification of the model. Figure 4 supplies a temporal

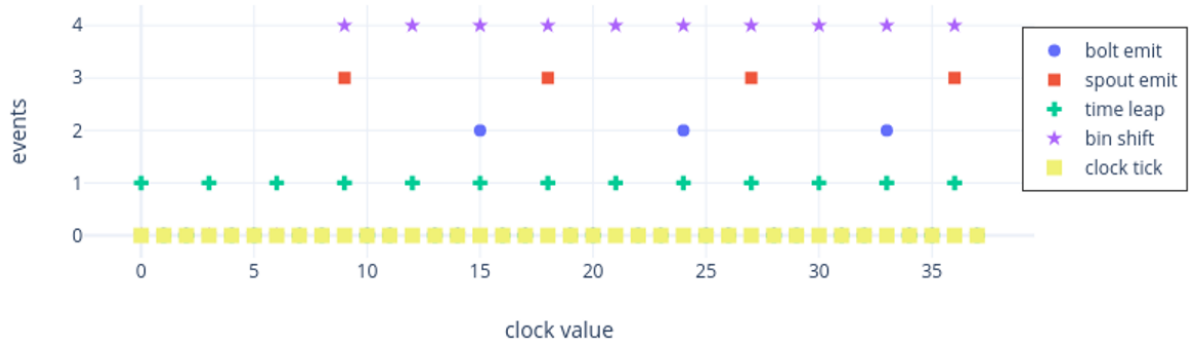


Figure 4: Time granularity.

diagram of the events: yellow squares represent time progress at each tick (event 0); green crosses represent time leaps (event 1); red squares represent the occurrence of spout emits (event 3); lilac stars represent a thread starting processing and its moves from one bin to the next (event 4); blue dots symbolise the emissions of threads, which occur following the issuances from the spout (along with the corresponding initiation of thread computation) by EB ticks (event 2). *No event occurs in between two time leaps.*

The time leap abstraction can be applied provided that two conditions are satisfied. The number of bins $b + 2$ calibrates the modelled time granularity. The conditions are:

- EB must be a multiple of b, i.e. $EB = \alpha \times b$ for some integer $\alpha \geq 1$
- ES must be a multiple of α.

5.2. Detailed Formalisation of the Models

Several topologies have been modelled in order to assess the scalability of the devised strategy (Section 6).[3] In this section, we describe in detail the general structure common to all models, which behave in the same manner and differ only in the connection pattern amongst nodes.

NuSMV models include one or more instances of (different) modules, each one capturing the behaviour of a component of the modelled reactive system. NuSMV modules are parametric and parameters are never modified by model transitions; parameters are instantiated when a module is operated by the

[3] All the models, results, software and scripts used for this research work are publicly available on a dedicated Zenodo repository [17].

verification tool. Model parameters are: for each bolt B_j, the thread cardinality `threads_in_Bj`, the upper bound on the queue length `queue_in_Bj`, and `emission_in_Bj` (corresponding to EB_j); for each spout S_i, its `emission_in_Si` (corresponding to ES_i); and `leap` which is the granularity of time progress.

The model is designed to represent the executions of Storm topologies cyclically in *rounds*, composed by six *steps* each counted by a global variable φ.

A NuSMV module is created for each spout S_i, having parameters: φ; TL_i, taking the value of `leap`; ES_i taking the value of `emission_in_Si`; and boolean parameters $b_{i.y}$, with $y \in [1, \mathcal{B}]$, indicating whether S_i has an outgoing edge towards B_y or not. Local variables for a spout module are: the local clock TS_i, and $emit_i \in [0, \mathcal{B}]$ that – if the spout can emit – indicates to which bolt it emits; if $emit_i = 0$ then the spout does not emit. The $emit_i$ variable introduces nondeterminism; the emission actually takes place only if the spout is connected to the bolt indicated by $emit_i$. In case of emission, the local variables $E_{i.y}$, with $y \in [1, \mathcal{B}]$, record the number of tuples (maximum 1) injected in B_y's queue from time to time.

A NuSMV module is created for each bolt B_j, having parameters: j; φ; integer parameters $rs_{j.x}$, with $x \in [1, \mathcal{S}]$, indicating the number of tuples that B_j is receiving from S_x;[4] integer parameters $rb_{j.x}$, with $x \in [1, \mathcal{B}]$, indicating the number of tuples that B_j is receiving from B_x; boolean parameters $eb_{j.x}$, with $x \in [1, \mathcal{B}]$, indicating whether B_j has an outgoing edge towards B_x or not; tb_j set to `emission_in_Bj`; $Nthreads_j$ set to `threads_in_Bj`; and $error_j$ set to `queue_in_Bj`. Without loss of generality and for the sake of greater clarity, let us assume that $b = 3$ bins are considered in the following explanation. Then, the local variables for a bolt module are: its current queue length Q_j; the counters TI_j, $T0_j$, $T33_j$, $T67_j$ and $T100_j$; integer variables $E_{j.x}$, with $x \in [1, \mathcal{B}]$, indicating the number of tuples (maximum 1) emitted to B_x's queue from time to time.[5]

The unsafe formula (see Section 3.1) is defined as:

$$\upsilon \;:=\; \bigvee_{j=1}^{\mathcal{B}} B_j.Q_j > \texttt{queue\_in\_Bj}$$

The initial state of a model is:

$$\begin{aligned}\iota \;:=\;& \varphi = 0 \wedge (\forall S_i.TS_i = 0 \wedge (\forall y.E_{i.y} = 0)) \wedge (\forall B_j.Q_j = 0 \wedge TI_j = Nthreads_j \\ & \wedge T0_j = 0 \wedge T33_j = 0 \wedge T67_j = 0 \wedge T100_j = 0 \wedge (\forall x.E_{j.x} = 0))\end{aligned}$$

In $\varphi = 0$, spouts possibly emit a tuple. This is described by the following transitions in the spout module of each S_i.

$$\begin{aligned}\tau_1 \;:=\;& \varphi = 0 \wedge TS_i \geq ES_i \wedge (\exists x.b_{i.x} \wedge emit_i = x \wedge TS_i' \leftarrow 0 \wedge E_{i.x}' \leftarrow 1 \wedge \varphi' \leftarrow 1) \\ \tau_2 \;:=\;& \varphi = 0 \wedge TS_i \geq ES_i \wedge (\forall x.\neg(b_{i.x} \wedge emit_i = x) \wedge TS_i' \leftarrow 0 \wedge E_{i.x}' \leftarrow 0 \wedge \\ & \varphi' \leftarrow 1)\end{aligned}$$

It is worth to notice that $emit_i$ is not initialised in ι, nor its value is ever determined by an assignment. As a consequence, τ_1 may be verified for different x's in different spouts, and for all of them the value of $E_{i.x}$ is updated as shown.

In $\varphi = 1$, the bolt module of each B_j updates its queue accordingly:

$$\tau_3 \;:=\; \varphi = 1 \wedge (Q_j + \sum_{n=1}^{\mathcal{S}} rs_{j.n}) \leq error_j \wedge Q_j' \leftarrow Q_j + \sum_{n=1}^{\mathcal{S}} rs_{j.n} \wedge \varphi' \leftarrow 2$$

$$\tau_4 \;:=\; \varphi = 1 \wedge (Q_j + \sum_{n=1}^{\mathcal{S}} rs_{j.n}) > error_j \wedge Q_j' \leftarrow error_j + 1 \wedge \varphi' \leftarrow 2$$

[4] Hence, $rs_{j.i}$ in B_j equals $E_{i.j}$ in S_i.
[5] Hence, $rb_{j.i}$ in B_j equals $E_{i.j}$ in B_i.

In $\varphi = 2$, threads in each B_j emit processed tuples and return idle:

$$\begin{aligned}\tau_5 &:= \varphi = 2 \wedge (\exists x.eb_{j.x} \wedge E'_{j.x} \leftarrow T100_j \wedge T100'_j \leftarrow 0 \wedge TI'_j \leftarrow TI_j + T100_j \wedge \\ &\quad \varphi' \leftarrow 3) \\ \tau_6 &:= \varphi = 2 \wedge (\forall x.\neg eb_{j.x} \wedge E'_{j.x} \leftarrow 0 \wedge T100'_j \leftarrow 0 \wedge TI'_j \leftarrow TI_j + T100_j \wedge \varphi' \leftarrow 3)\end{aligned}$$

In τ_5, all threads that have finished processing emit toward the same downstream bolt; this is due to the fact that, in the topologies considered in this work, each bolt emits towards at most another one. Transition τ_6 is performed by terminal bolts whose processing results exit the system.

In $\varphi = 3$, the queue of each B_j is updated accordingly.

$$\begin{aligned}\tau_7 &:= \varphi = 3 \wedge (Q_j + \sum_{n=1}^{\mathcal{B}} rb_{j.n}) \leq error_j \wedge Q'_j \leftarrow Q_j + \sum_{n=1}^{\mathcal{B}} rb_{j.n} \wedge \varphi' \leftarrow 4 \\ \tau_8 &:= \varphi = 3 \wedge (Q_j + \sum_{n=1}^{\mathcal{B}} rb_{j.n}) > error_j \wedge Q'_j \leftarrow error_j + 1 \wedge \varphi' \leftarrow 4\end{aligned}$$

In $\varphi = 4$, idle threads of each B_j take in charge queued tuples according to the pseudo-code supplied in Section 5.

$$\begin{aligned}\tau_9 &:= \varphi = 4 \wedge Q_j > 0 \wedge TI_j \geq Q_j \wedge Q'_j \leftarrow 0 \wedge T0'_j \leftarrow T0_j + Q_j \wedge \\ &\quad TI'_j \leftarrow TI_j - Q_j \wedge \varphi' \leftarrow 5 \\ \tau_{10} &:= \varphi = 4 \wedge Q_j > 0 \wedge TI_j < Q_j \wedge Q'_j \leftarrow Q_j - TI_j \wedge T0'_j \leftarrow T0_j + TI_j \wedge \\ &\quad TI'_j \leftarrow 0 \wedge \varphi' \leftarrow 5\end{aligned}$$

Finally, **in $\varphi = 5$, for each spout S_i its timer TS_i is incremented of one time** `leap` (reset is in step $\varphi = 0$), **and bin shifts take place in bolts**.

$$\begin{aligned}\tau_{11} &:= \varphi = 5 \wedge (\forall S_i.TS_i < ES_i \wedge TS'_i \leftarrow TS_i + TL_i) \wedge (\forall B_j.T100'_j \leftarrow T67_j \wedge \\ &\quad T67'_j \leftarrow T33_j \wedge T33'_j \leftarrow T0_j \wedge T0'_j \leftarrow 0) \wedge \varphi' \leftarrow 0 \\ \tau_{12} &:= \varphi = 5 \wedge (\forall S_i.TS_i \geq ES_i \wedge TS'_i \leftarrow TS_i) \wedge (\forall B_j.T100'_j \leftarrow T67_j \wedge \\ &\quad T67'_j \leftarrow T33_j \wedge T33'_j \leftarrow T0_j \wedge T0'_j \leftarrow 0) \wedge \varphi' \leftarrow 0\end{aligned}$$

6. Experimental Results

Storm topologies involve one or more of the basic relationships between nodes, namely, series connection, output to more than one other node (divergence), input from more than one other node (convergence). Figure 5 shows the considered models – of increasing complexity – which all include gradually larger parts of the reference topology in Figure 1(*a*), indicated as *PoliMI* in the following. The process of converting a Storm topology into a NuSMV model is performed *automatically*: we implemented a software that takes in input the nodes parameters and the description of their connections, and generates a NuSMV model according to the schema in Section 5.2. The software is available via [17].

The experiments have been run in a system with Windows 10 Pro OS, equipped with Intel Core i7-10510U 1.80 GHz processor and 16GB RAM. NuSMV 2.6.0 has been used, linked to the CUDD library version 2.4.1. NuSMV used BDD-based symbolic model checking, and LTL logic. Performance is obtained using the `print_usage` and `print_bdd_stats` NuSMV commands.

Table 1 summarises some tested configurations and the obtained results, with 3 bins. When more values are supplied for either ES or the number of threads N between square brackets, values listed therein refer – in order – to the various spouts and bolts respectively in the considered topology. Otherwise, all spouts and bolts in the topology are homogeneous. When $EB = 3$ ($\alpha = 1$), the time leap mechanism is not used (recall that $EB/\alpha + 2$ is the total number of bins used to model the

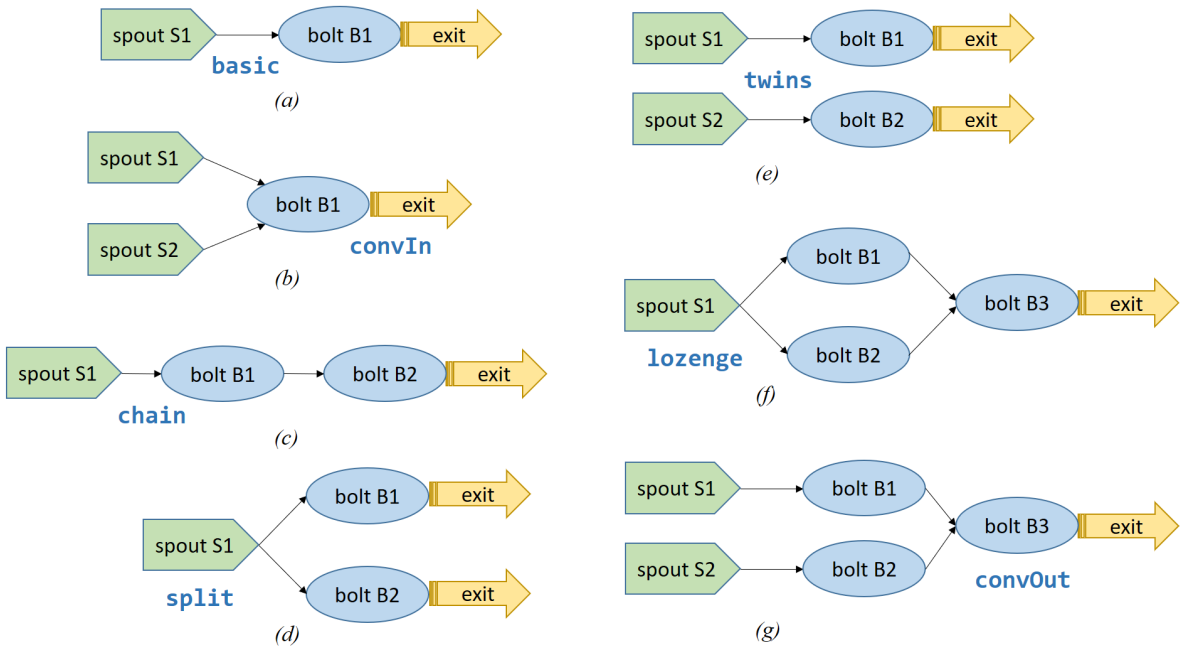


Figure 5: Tested Storm topologies.

computation of bolts). The *nodes* column is the number of BDD nodes allocated; *FSM* $\varnothing$ is the Finite State Machine diameter; *init* is the number of BDD nodes representing the init set of states; *vars* is the number of BDD variables; *cex-states* is the length of the counterexample trace. Two unsafe formulæ are considered for the *convOut* and the *PoliMI* models. For the former, either $\bigvee_{j=1}^{\mathcal{B}} B_j.Q_j > 1$ (A) or $B_1.Q_1 > 1 \vee B_2.Q_2 > 1 \vee B_3.Q_3 > 2$ (B). For the latter, either $B_1.Q_1 > 1 \vee B_2.Q_2 > 2 \vee B_3.Q_3 > 1$ (C) or $B_1.Q_1 > 1 \vee B_2.Q_2 > 2 \vee B_3.Q_3 > 2$ (D).

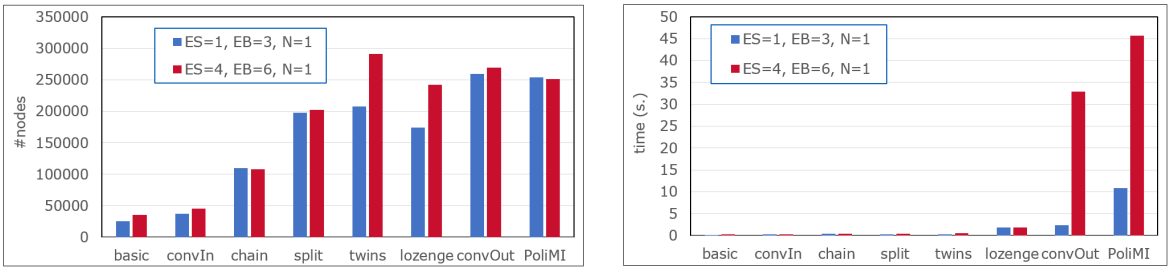
The following observations emerge from Table 1. First: the more complex the topology, the more expensive its verification. However, as far as safe configurations are concerned, while the complexity of the last three topologies may affect the number of BDD nodes generated – in particular for higher ES, EB and N – it does not significantly affect the latency for obtaining a result, which always amounts to a few tens of ms. Second, unsafe conditions (A) for *convOut* and (C) for *PoliMI* are unsafe even for high ES and high N because, for B_3, they violate the observation in Sec.3.1 about the link between fan-in of a bolt and lower bound on its queue limit. Third, checking an unsafe configuration requires far higher time and number of BDD nodes than safe configurations, for all topologies. Fourth, for unsafe configurations, increasing N causes a relevant increase in both time and number of nodes; observe for instance the case of the *convIn* topology, or the first two configurations for *chain*.

Four plots have been extracted from the complete results available in [17]. Figure 6 compares the topologies for two configurations, for which all the models are unsafe; the unsafe formulæ (B) and (D) have been used for *convOut* and *PoliMI* respectively. While the number of nodes (Fig.6(a)) is not heavily affected by increasing ES and EB, apart from *twins* and *lozenge*, the verification time (Fig.6(b)) for both *convOut* and *PoliMI* increases remarkably with respect to the other models, and this trend is emphasised for increasing ES and EB because a larger FSM must be analysed by NuSMV. With $ES = 1$ and $EB = 3$, the FSM diameter for both *convOut* and *PoliMI* is 94; with $ES = 4$ and $EB = 6$ the FSM diameter becomes 134 and 146 respectively. This observation confirms the state explosion causing part of the problems encountered in [6].

Figure 7(a) shows the verification time required to analyse both safe and unsafe *PoliMI* topologies, with $ES = 1$ and $EB = 3$ and variable number of threads in the bolts (shown between squared brackets on the x-axis), and unsafe formula (D). The first 5 configurations are all unsafe and yielded by

Table 1
Experimental results with $k = 3$ bins

topology	ES	EB	α	N	Q	result	time	FSM ∅	init	nodes	vars	cex-states
basic	1	3	1	2	> 1	unsafe	0.17s	44	38	31141	79	46
basic	1	3	1	6	> 1	SAFE	0.01	27	37	7153	79	
basic	6	3	1	1	> 1	SAFE	0.01	57	38	7036	79	
basic	4	6	2	2	> 1	SAFE	0.01	33	38	6851	79	
basic	6	18	6	2	> 1	unsafe	0.18	44	38	31944	79	46
basic	16	12	4	1	> 1	SAFE	0.01	45	38	6853	79	
convIn	1	3	1	2	> 2	unsafe	0.18	38	44	54836	93	40
convIn	1	3	1	3	> 2	unsafe	0.21	38	44	62151	93	40
convIn	1	3	1	5	> 2	unsafe	0.23	44	43	66287	79	46
convIn	1	3	1	6	> 2	SAFE	0.01	27	43	10909	93	
convIn	4	6	2	2	> 2	unsafe	0.34	86	44	80085	93	88
convIn	[4, 2]	6	2	5	> 2	SAFE	0.01	33	43	10554	93	
convIn	16	12	4	1	> 2	unsafe	0.47	146	44	73871	93	148
chain	1	3	1	2	> 1	unsafe	0.35	45	75	149540	155	70
chain	1	3	1	[6, 2]	> 1	unsafe	0.56	64	74	221916	155	84
chain	9	3	1	1	> 1	SAFE	0.02	93	75	32704	155	
chain	4	6	2	2	> 1	SAFE	0.02	51	75	18959	155	
chain	14	6	2	1	> 1	SAFE	0.02	81	75	17181	155	
split	1	3	1	[6,2]	> 1	unsafe	0.28	63	74	235347	155	46
split	9	3	1	1	> 1	SAFE	0.02	75	75	15994	155	
split	4	6	2	[1, 2]	> 1	unsafe	0.33	105	75	126165	155	88
split	14	6	2	1	> 1	SAFE	0.02	63	75	15783	155	
twins	1	3	1	2	> 1	unsafe	0.33	46	82	266875	173	46
twins	2	3	1	2	> 1	SAFE	0.02	33	82	19065	173	
twins	[2, 6]	6	2	1	> 1	unsafe	0.21	50	82	120995	173	40
lozenge	1	3	1	2	> 1	unsafe	2.19	88	120	187458	245	70
lozenge	1	3	1	[6, 6, 3]	> 1	SAFE	0.05	45	118	63799	245	
lozenge	6	9	3	[1, 1, 2]	> 1	unsafe	1.14	141	120	836948	245	88
lozenge	6	9	3	[2, 2, 1]	> 1	unsafe	0.86	106	120	543874	245	108
lozenge	16	12	4	1	> 1	SAFE	0.04	63	120	30192	245	
convOut	1	3	1	[3, 3, 4]	(B)	unsafe	36.42	64	127	289022	265	90
convOut	1	3	1	[3, 3, 6]	(B)	SAFE	0.06	64	127	88621	265	
convOut	16	6	2	8	(A)	unsafe	1.31	87	128	478281	265	136
convOut	16	6	2	1	(B)	SAFE	0.07	105	128	42514	265	
PoliMI	6	9	3	[2, 3, 3]	(D)	SAFE	0.08	69	128	124832	265	
PoliMI	16	6	2	8	(C)	unsafe	1.33	87	128	506554	265	136
PoliMI	16	6	2	1	(D)	SAFE	0.06	105	128	45731	265	

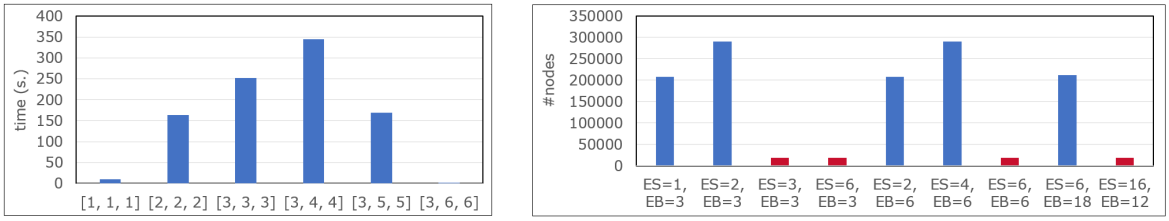


(a) (b)

Figure 6: Comparison among topologies in terms of (a) number of BDD nodes, and (b) time spent for the verification.

both spouts emitting towards B_2; though, with 3 threads in B_1 and 5 threads in B_2 and B_3 each, the counterexample shows that the queue overflow in B_2 is quickly overcome, leading to an oscillatory behaviour. The configuration [3, 6, 6] is safe, and the time spent drastically drops to 0.30s. In parallel, the number of nodes increases from 253389 of the configuration [1, 1, 1], up to 380669 of the configuration [3, 3, 3], and 546089 of the configuration [3, 6, 6]. The FSM diameter grows from 94 of the configuration

[1, 1, 1], to 100 for the configuration [3, 4, 4], and drops to 45 for the configuration [3, 6, 6]. The



(a) (b)

Figure 7: (a) Time spent for the verification of the *PoliMI* topology with $ES = 1$, $EB = 3$, variable number of threads in the bolts, and unsafe condition (D). (b) Number of BDD nodes for the *twins* topology for $Nthreads = 1$ and variable ES and EB.

configuration [3, 4, 4] is the worst one over all our experiments in terms of computation time.

In Figure 7(b), the behaviour of the *twins* topology is analysed for either safe (red) or unsafe (blue) configurations, with 1 thread on each bolt and variable ES and EB. It is evident the overhead generated by NuSMV to identify unsafe configurations and compute counterexamples.

Increasing the number of bins reproduces the evolution of tuple processing by threads with finer granularity, thus ideally producing "more realistic" models. However, it asymptotically tends towards a model that re-introduces the convergence problem observed in [6], confirming the usefulness of the technique proposed in this work. We performed experiments for PoliMI, with the same settings of threads per bolt as in Figure 7(a), but considering 4 bins and $EB = 4$ (with $ES = 1$ always). The computation time with 2 threads per bolt was 468.11s; with 3 threads per bolt it increased to 2184.61s, while the three successive configurations – all unsafe – were checked in 4841.99s, 5697.7s and 7120.99s respectively. The number of BDD nodes amounts to 2,239,577 with 3 threads per bolt, and to 6,405,418 in the last case. Yet, a configuration with [4, 8, 8] threads in the bolts is proved safe in 2.75s.

Finally, the *funnel effect* (topology in Fig.2) was also reproduced successfully. With $ES = 20$, $EB = 3$, 1 thread per bolt, and unsafe condition $B_1.Q > 3 \vee B_2.Q > 1$, NuSMV supplies a *safe* result; if the number of threads in B_1 is increased to 3 then the result becomes *unsafe*.

7. Conclusions

In this work, a strategy to model Storm topologies and perform their formal verification as a finite-state problem through unbounded model checking is presented. The results obtained through experiments conducted using NuSMV show that the strategy scales well, and all the models and configurations analysed have been successfully verified in acceptable times, and without memory exhaustion.

We are investigating the application of the same strategy to the infinite-state problem (using NuXMV), which would enable designing topologies that are safe for any number of threads above a certain threshold, and preliminary results are promising in terms of performance. To make it possible, we are working to single out and exclude conditions of system oscillations possibly implicated by the funnel effect. Furthermore, we will extend our modelling strategy by considering topologies where the fan-out of bolts can be > 1 and emitted tuples can distribute non-deterministically in any way to downstream bolts. This is simply an extension of the software, which presents no conceptual obstacles.

Declaration of Generative AI and AI-assisted Technologies

During the preparation of this work, the authors used ChatGPT for grammatical and spelling check, paraphrasing and rewording, and to improve the writing style of the abstract only. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Universal Guards, Relativization of Quantifiers, and Failure Models in Model Checking Modulo Theories. Satisfiability, Boolean Modeling and Computation **8**(1-2), 29-61 (2012).

[2] Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183-235 (1994).

[3] Apache Software Foundation, Apache Storm Tutorial version 2.8.6, https://storm.apache.org/releases/2.8.6/Tutorial.html. Last accessed 24 Apr. 2026.

[4] Apache Software Foundation, Companies Using Apache Storm, https://storm.apache.org/Powered-By.html. Last accessed 24 Apr. 2026.

[5] Apache Software Foundation, Performance Tuning, https://storm.apache.org/releases/2.8.6/Performance.html. Last accessed 24 Apr. 2026.

[6] Bersani, M.M., Marconi, F., Rossi, M., Erascu, M., Ghilardi, S.: Formal Verification of Data-Intensive Applications through Model Checking Modulo Theories. In: Proc. 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN), pp. 98-101. ACM, New York (2017).

[7] Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems. In: Parthasarathy, M., Seshia, S.A. (eds), Computer Aided Verification, LNCS, vol. 7358, pp. 718-724, Springer Heidelberg (2012).

[8] De Moura, L.M., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds) Tools and Algorithms for Construction and Analysis of Systems, LNCS, vol. 4963, pp. 337-340, Springer Heidelberg (2008).

[9] Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: Giesl, J., Hähnle, R. (eds) Automated Reasoning, LNCS, vol. 6173, pp.22-29. Springer, Heidelberg (2010).

[10] Karbyshev, A., Bjørner, N.S., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-Directed Inference of Universal Invariants or Proving Their Absence. Journal of the ACM **64**(1), 1-33 (2017).

[11] Kirner, T.G.: Quality Requirements for Real-Time Safety-Critical Systems. IFAC Proceedings Volumes **29**(5), 103-108 (1996).

[12] Lestingi, L., Sbrolli, C., Scarmozzino, P., Romeo, G., Bersani, M.M., Rossi, M.: Formal modeling and verification of multi-robot interactive scenarios in service settings. In: Proc. IEEE/ACM 10th International Conference on Formal Methods in Software Engineering, pp. 80-90. ACM, New York (2022).

[13] Marconi, F., Bersani, M.M., Erascu, M., Rossi, M.: Towards the Formal Verification of Data-Intensive Applications Through Metric Temporal Logic. In: Ogata, K., Lawford, M., Liu, S. (eds) Formal Methods and Software Engineering, LNCS, vol. 10009, pp. 193-209. Springer, Cham (2016).

[14] Marconi, F., Bersani, M.M., Rossi, M.: A model-driven approach for the formal verification of storm-based streaming applications. ACM SIGAPP Applied Computing Review **17**(3), 6-15 (2017).

[15] Marconi, F., Bersani, M.M., Rossi, M.: Formal verification of storm topologies through D-VerT. In: Proc. Symposium on Applied Computing, pp. 1168–1174. ACM, New York (2017).

[16] NuSMV Homepage, https://nusmv.fbk.eu/index.html. Last accessed 24 Apr. 2026.

[17] Pagani, E., Bersani, M.M., Erascu, M., Marconi, F., Ghilardi, S.: Formal verification of Storm topologies - Supplementary Material. Zenodo (2024). https://doi.org/10.5281/zenodo.10955819. Last accessed 24 Apr. 2026.

[18] Qadir, J., Hasan, O.: Applying Formal Methods to Networking: Theory, Techniques, and Applications. IEEE Communications Surveys & Tutorials **17**(1), 256-291 (2015).

[19] Souri, A., Navimipour, N.J., Rahmani, A.M.: Formal verification approaches and standards in the cloud computing: A comprehensive and systematic review. Computer Standards & Interfaces **58**, 1-22 (2018).

[20] Souri, A., Norouzi, M., Asghari, P., Rahmani, A.M., Emadi, G.: A systematic literature review on formal verification of software-defined networks. Wiley Trans. Emerging Telecommunications Technologies **31**(2) (2020).

[21] The Zot bounded satisfiability checker. https://github.com/fm-polimi/zot. Last accessed 24 Apr.

2026.
[22] Yao, J., Xu, C., Li, D., Lin, S., Cao, X.: Formal Verification of Security Protocols: ProVerif and Extensions. In: Sun, X., Zhang, X., Xia, Z., Bertino, E. (eds) Artificial Intelligence and Security, LNCS, vol. 13339, pp. 500-512. Springer, Cham (2022).