Formal Verification of Data-Intensive Applications through Model Checking Modulo Theories

Marcello M. Bersani, Francesco Marconi, Matteo Rossi
DEIB, Politecnico di Milano
{firstname.lastname}@polimi.it

Madalina Erascu
Institute e-Austria Timisoara & West University of Timisoara, Timisoara, Romania
merascu@info.uvt.ro

Silvio Ghilardi
Università degli Studi di Milano
silvio.ghilardi@unimi.it

ABSTRACT

We present our efforts on the formalization and automated formal verification of data-intensive applications based on the Storm technology, a well known and pioneering framework for developing streaming applications. The approach is based on the so-called *array-based systems* formalism, introduced by Ghilardi et al., a suitable abstraction of infinite-state systems that we used to model the runtime behavior of Storm-based applications. The formalization consists of quantified formulae belonging to a certain fragment of first-order logic to symbolically represent array-based systems. The verification consists in checking whether some safety property holds or not for the system. Both formalization and verification are performed in the same framework, namely the state-of-the-art Cubicle model checker.

CCS Concepts

•Software and its engineering → Formal software verification;

Keywords

Data-intensive applications; Storm technology; Formal verification; Array-based systems; infinite-state model checking.

1. INTRODUCTION

Data-intensive applications (DIAs) are able to process large volumes of data—the so-called Big Data—in a reasonable amount of time. Big Data has gained a lot of attention in the last years because of the fast-growing volumes of information manipulated by applications.Hence, it is of foremost importance that developers can rely on innovative solutions to support the entire DIAs life-cycle. Our work is part of the European DICE project [3] which seeks to define methods and tools for the data-aware quality-driven development of DIAs. We focus on the Apache Storm (storm.apache.org) technology widely used in applications that need efficient processing of unbounded streams of data, such as event log monitoring, real-time data analytics and data normalization.

Our long term goal is the representation, at design-time, of the runtime behavior of Storm topologies. A *topology* provides an abstract representation of a DIA through directed graphs, where nodes are of two kinds: *computational nodes* (named bolts in Storm) implement the logic of the application by elaborating information and producing an outcome, whereas *input nodes* (named spouts in Storm) bring information into the application from the environment. The aim is avoiding as much as possible the possibility that applications need a re-design after their deployment. To this end, we evaluate the applicability of the array-based systems approach [4] implemented in state-of-the-art tools MCMT[1] and Cubicle[2] to the modeling of Storm applications. In particular, in this paper we focus on the *safety* analysis of Storm topologies. Safety verification aims to verify that undesired behaviors, or configurations, will not occur in the system.

The *first contribution* of the work is the first application of array-based systems to model and verify parametric aspects of DIAs in an infinite-state paradigm. Array-based systems have been successfully applied for the formalization and verification of parameterized timed systems, fault tolerant systems and imperative programs, but never to DIAs. The *second contribution* is the understanding that the lack of a proper abstraction to model arbitrary (yet finite) processes in Storm components impedes the identification of a meaningful verification problem that can be defined in terms of safety verification of an array-based system. The *third contribution* is a set of lessons learned concerning the state-of-the-art tools, which do not support specific features for limiting the state space exploration with user-defined criteria that would promote a reduction of time and memory required for the analysis. In fact, the last model was obtained after many steps of refinement needed to overcome the limitations of the tools (see the analysis in Sect. 4.1).

We approached a similar problem in [5], but with a different formalism, namely the CLTLoc temporal logic [2].The verification in [5] was limited to applications with a *bounded number of threads* running in bolts. The model introduced in this paper must be considered as a complement to the temporal one, given its ability to represent an *arbitrary number of processes* (parameterized infinite-state model checking). This feature can be used to analyze an arbitrary level of parallelism in DIA components such as Storm bolts. In the CLTLoc model the length of the queue associated with a bolt might become infinite if the bolt cannot timely process

[1]http://users.mat.unimi.it/users/ghilardi/mcmt/
[2]http://cubicle.lri.fr/

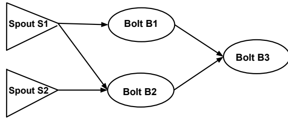

Figure 1: **Example of Storm topology.**

the incoming stream. In [5], we verified the property that "all bolt queues cannot grow unboundedly". In this work, the verification problem is a *coverability analysis*, checking whether, given a queue(s) bound(s) defined by the designer, "all the bolt queues have a limited occupation level".

2. PRELIMINARIES

In this section we provide a brief overview of the Apache Storm technology and of the formalism chosen to model applications based on this technology.

Apache Storm 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 string-based messages, called 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*. Spouts are stream sources. They generally get data from external systems such as queuing brokers or web APIs. Bolts apply transformations over the incoming data streams and generate new output streams to be processed by the connected bolts. Connections are defined at design time by the subscription of the bolts to spouts or other bolts. The number of processes running in parallel for each node can be arbitrary (unbounded *parallelism*). Figure 1 shows an example of Storm topology that will be used in Section 4. It is composed of 2 spouts and 3 bolts: B_1 subscribes to the tuples emitted by S_1, B_2 to those emitted by S_1 and S_2, and B_3 to those emitted by bolts B_1 and B_2.

Array-based Systems [4] are the mechanism we exploit to formalize the behavior of Storm topologies, as their state can be seen as a set of unbounded arrays whose indexes range over elements of the parameterized domain. The parametrization is on the number of processes of the topology nodes as the number of nodes in a Storm application is fixed and never changes.

An array-based system is a tuple $S = (A, Init, \tau)$ where A is a set of function symbols (arrays) representing the state variables, $Init$ is a formula which characterizes the initial states of the system (in which the variables from A can appear free) and τ is a transition relation. Transition relation τ is expressed as a disjunction of existentially quantified (by zero, one, or several variables of the parameterized domain) formulae, where each disjunct is a parameterized transition of the system. A transition t relates the array variable a with an updated variable a' and has the form $\underset{\bar{i}}{\exists}\, G(\bar{i}, A) \wedge \bigwedge_{a \in A} \underset{\bar{j}}{\forall} a'(\bar{j}) = U_a(\bar{i}, \bar{j}, A)$. G is called the guard of t, U_a the update of a, and the conjunction of the formulae following G is the action of the transition t. Safety properties are expressed by characterizing unsafe states. An unsafe formula is a conjunction of literals l_k, $k = \overline{1,n}$, existentially quantified: $\underset{\bar{i}}{\exists}\, l_1(\bar{i}) \wedge ... \wedge l_n(\bar{i})$. The formalization of an array-based system consists of the set of initial states, the ordering of the actions (by means of a transition relation) and the set of unsafe states.

Array-based systems can be formally verified through a decision procedure based on *backward reachability*. Backward reachability analysis is based on the idea of repeatedly computing the pre-image of the set of unsafe states (obtained by complementing the property to be verified) and checking for fix-point and emptiness of the intersection with the set of initial states. This technique can be used to analyze parameterized systems consisting of a finite (but unknown) number n of identical processes modeled as extended finite state automata, which manipulate variables whose domains can be unbounded, like integers. The *challenge*, which holds for parameterized systems in general and for DIAs in particular, is to check safety properties for any number n of processes.

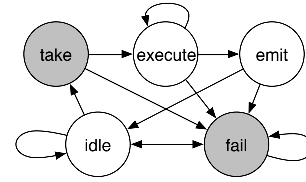

Figure 2: **Automaton describing a bolt's behavior.**

3. MODELING ASSUMPTIONS

This section describes the basic principles underlying our model of Storm topologies. The model focuses on the behavior of the queues of the bolts and 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. To this end, our models capture the *progress of time* in the system and make use of discrete counters to describe the evolution of the *size of the queues*.

Some of the modeling assumptions are the same as in [5]. We do not consider deployment details, such as the number of worker processes and the underlying architecture. Spouts do not have any input queue or incoming connection from other bolts, whereas bolts have internal queues, of unbounded size, which store the incoming tuples received from subscribed nodes of the topology. The various modeling exercises in this work used two different ways for abstracting bolts: one where each bolt has one receiving queue for each of its parallel instances, and another where only one single receiving queue is shared among all its parallel instances. In any case, no sending queue is represented. We do not detail the contents of tuples, but only their quantities, since the verification problem is on the size (number of tuples) of the queues. Spouts are considered sources of tuples and their queues are not represented.

A Storm topology is a directed graph where the set of nodes includes the sets of spouts and bolts. Our models represent a topology by encoding the subscription relation among bolts and spouts.

The behavior of bolts can be illustrated by means of a finite state automaton (see Figure 2, taken from [5]). However, as discussed in Section 4.1, we had to simplify the automaton and remove state *take*. Moreover, we do not capture bolt failures, so state *fail* is not part of the model. Spout failures are also not captured, because their effect is irrelevant for the growth analysis of bolt queues as they would reduce the workload on the topology.

The timing parameters of a topology are: a) the time required by bolts to process a tuple (which is called *Execution rate*); b) the minimum time between two consecutive spout emits. We do not consider the maximum time because we can assume that a topology with sparse spout emits (with emit time tending towards infinity) is safe. We model one-to-one communication (in Figure 2, when S_1 emits the tuple is sent either to B_1 or to B_2) to write simpler formulae but we can easily extend the model to multicast.

4. FORMALIZATION AND VERIFICATION

This section outlines our formal model capturing the topology of Figure 1 with the features presented in Section 3. The final model was obtained after many refinements devised to overcome the challenges described in Sect. 4.1.

The fundamental aspects that characterize a topology and its associated array variables used in the models are described in the following (when not specified, the dimension of the array has to be considered not constant in all the models we implemented, as it varied from a version to another). The duration of the entire processing is kept in the real variable T representing the global clock. The status of the bolt with index i of process j is indicated by variable $B[i,j]$, which can be equal to: (`E`)*mit*, (`I`)*dle* and *E*(`X`)*ecute* (*state Ta*(`K`)*e* was initially considered but omitted in the end). The length of the queue associated to the bolt is indicated by an array variable L. The number of tuples that are being elaborated by process j of bolt i, since the last take, is indicated by variable $P[i,j]$. The elapsed time from the last emit action performed by spout i is indicated by an array variable *Stime*. The constant Ts_{min} is a parameter for the system and represents the minimum time between two consecutive spout emit performed by the same bolt. The subscription relation among bolts and spouts and among bolts are indicated with the array variables *SubscribedBS* and *SubscribedBB* of booleans, respectively. Predicate $SubscribedBS[i,j]$ indicates that "bolt i subscribes to the streams emitted by spout j" while $SubscribedBB[i,j]$ indicates that "bolt i subscribes to the streams emitted by bolt j".

Verification problem. The safety property we are interested in requires that all the bolts have bounded queue, i.e., the size of their queue does not overflow a given constant. To verify the property we check if a state satisfying the negation of the property can be reached from the initial state. This formula encoding the property varies from model to model as it depends on the queue modeling: in the case of *shared queue* all the active processes in a bolt use the same queue, while in the case of *multiple queue* each active process in a bolt uses a separate queue. The formula for the *shared queue* case is of the form $\underset{i}{\forall} L[i] < c$, with c a constant value.

Modeling topologies. We chose to model the evolution of a topology with both discrete and time elapse transitions, similarly to the standard semantics of Timed Automata [1]. This allows us to differentiate the effect of the transitions, as they are fired in an alternating manner, and to simplify the model, as discrete transitions only enforce updates on state variables and not on the time T (the time elapsing affects the processing of tuples carried out by bolts). Discrete transitions have the purpose of either changing the state of the components or updating the size of the queues of the bolts but they do not modify the value of variable T. Conversely, the transition modeling the elapsing of time adds a positive amount δ to variable T. It possibly changes the states of some bolts when their processing has been terminated during the last δ time units. To model the alternation between the two types of transitions, we include in the model a flag called *CanTimeElapse* that, when positive, it allows time elapsing transition to fire.

For representing the graph of a topology we use invariants on the predicates *SubscribedBS* and *SubscribedBB*, as their value never change over the execution of the topology. We use formulae of the form

$$\underset{i,j}{\forall}\; i = 1 \land j \geq 2 \;\Rightarrow\; SubscribedBB[i,j] = \texttt{false}$$

The formula states that the bolt $B[1,x]$ is not connected to any other spout with index greater than or equal to 2 (based on Figure 1).

Time elapsing in the topology is expressed by (1). It exploits an array variable, called N_{proc}, which defines the number of active processes in bolt i. The value of $N_{proc}[i]$, for any i representing a bolt in the topology, is not fixed *a priori* as it is an arbitrary parameter of the system. $P[i,x]$ stores the percentage of the tuple which still has to be processed. When process x receives a tuple and starts the execution, the value of $P[i,x]$ is set to 1 and keeps decreasing from 1 to 0 while time progresses due to (1). Formula (1) states that if there exists a positive value δ and $canTimeElapse = \texttt{true}$ then: *(a)* the time progresses by incrementing T with δ time units; *(b)* for all the j and z such that δ is not big enough to complete the remaining part of the computation, i.e., $P[j,z] - \delta > 0$, then $P[j,z]$ is updated with the progress $P[j,z] - \delta$; otherwise, if δ allows the bolt to complete the current processing (when $P[j,z] - \delta < 0$) then $P[j,z]$ is set to 0. Observe that $P[j,z]$ is always ≥ 0 (by construction) and when it is non null then process z of bolt j is in execute state `X` and z is an index of an active process of the bolt i, i.e., $0 \leq z < N_{proc}[j]$.

$$\underset{\delta}{\exists} 0 < \delta \land canTimeElapse = \texttt{true} \land \underset{j,z}{\forall} \begin{pmatrix} T' & = T + \delta \\ P'[j,z] & = \textbf{if } (0 \leq P[j,z] - \delta) \\ & \quad \textbf{then } P[j,z] - \delta \textbf{ else } 0 \\ B'[j,z] & \ldots \\ CanTimeElapse' & = \texttt{false} \end{pmatrix} \qquad (1)$$

Various refinements were undertaken to obtain the essential core set of transitions modeling the topology behavior. However, the rationale behind the following transitions is common to all versions of the model we have devised. We describe the three of them without details by outlining their functionality only. The set of transitions describing the behavior of the Storm topology allows state change according to Figure 2 and how this affects the accumulation of tuples in the queues: (1) $spout_{emit}(i,j)$: the queue of the bolt j subscribed to the spout i increases and the emit time of the spout is reset, (2) $bolt_{emit}(i,j,x)$: the state of bolt i is changed into idle and the length $L[j]$ of the queue of bolt j is incremented by 1, (3) $bolt_{take}(j,y)$: the length $L[j]$ of the queue of bolt j is decreased by 1 and the percentage of processing of the thread receiving the tuple $P[j,x]$ is set to 1. It is worthy to observe that we did not consider the state `K` but we actually model the take action through $bolt_{take}$.

4.1 Challenges

We elaborate on the main challenges that we faced with writing the transition system modeling the topology.

Nondeterministic updates. The following transition captures the state change of spouts and bolts in one of its first versions (hence, it makes use of a boolean variable *statechange* and state `K` that were deprecated in following refinements of the model). It should be read as: if there exist the variables x, y, i, j such that the system is ready to change the state ($statechange = \texttt{true}$) then (1) in the next time instance the system can not change the state again ($statechange' = \texttt{false}$)[3] (2) the bolt $B[j,y]$ in the emit state can go into idle or take state or the bolt $B[j,y]$ in the idle state can go into state take, and (3) in the next time instances the system time elapses.

$$\underset{x,y,i,j}{\exists} statechange = \texttt{true} \land \underset{l,k,z}{\forall} \begin{pmatrix} statechange' = & \texttt{false} \land \\ \ldots & \\ B'[l,z] = & \textbf{if } (z = y \land l = j \land B[j,y] = \texttt{E}) \\ & \textbf{then } (\texttt{I} \textbf{ or } \texttt{K}) \textbf{ else } B[l,z] \\ & \textbf{elseif } \ldots \\ canTimeElapse' = \texttt{true} & \end{pmatrix} \qquad (2)$$

[3]Essentially, `statechange` can become `false` when the state has just been changed and now the time must elapse in order tuples to be processed.

When implementing the set of transitions of the system in Cubicle, we had to overcome some limitations of the tool which cannot handle nondeterministic updates that are built through formulae of the form similar to (`I` **or** `K`). For example, the transition above had to be split into various transitions to capture alternate states (or) and **if...elseif...** constructs, having all the same guard but implementing different deterministic updates.

Dimension of the state space. A second major issue we had to overcome is posed by the size of the state space that our models entail. In many cases, we experimented that the state space exploration is so huge that the termination of the backward reachability algorithm is not reached in reasonable time. Hence, we had to devise the following strategy to reduce the state space. (1) Different possible states for the spouts are left out; the model keeps track only of the time elapsing to enable spout emit. Therefore, *Stime* is the unique mono-dimensional array to represent spout functionality in the model. (2) Bolt queues have only one dimension, i.e., there is one queue for each bolt meaning that processes in a bolt share the same queue. For instance, these two restrictions allowed us to define simpler transitions, such as the one modeling the spout emit event (3).

$$\exists_{i,j,x} Ts_{min} < Stime[i] \wedge SubscribedBS[j,i] = \texttt{true} \wedge \ldots$$
$$\forall_{l} \begin{pmatrix} L'[l] & = \textbf{if } (l{=}j) \textbf{ then } L[l]{+}1 \textbf{ else } L[l] \\ Stime'[l] & = \textbf{if } (l{=}i) \textbf{ then } 0 \textbf{ else } Stime[l] \\ CanTimeElapse' & = \texttt{true} \\ \ldots & \end{pmatrix} \quad (3)$$

Incorrect firing of transitions. This issue stems from a limitation in the current state-of-the-art tools. The backward reachability algorithm that they implement does not allow the specification of the so-called urgent transitions. In timed automata verification, for instance, some tools implement this feature; namely, if an urgent transition is enabled then only that transition is fired first and all the other that are enabled are fired later. This way, the firing of some transitions could not be postponed. In our case, if a bolt is able to take tuples then it must take, otherwise its queue can easily become full without having the chance of being decreased or emptied. Also, if a bolt can emit it must emit to allow another tuple for processing. Therefore, the actions *take* and *emit* of bolts are considered urgent with respect to spout emits. To realize urgent transitions we made use of suitable boolean flags whose value is changed through specific transitions called flag transitions.

Number of transitions. Another issue we had to consider is the number of transitions in the model, that affect the time and the memory required to carry out the state space exploration. The reason depends on how the backward reachability algorithm is performed. Informally, given a configuration c of the system and a transition t in the model, the algorithm applies t on c to obtain the pre-image c' of c such that c is reachable in one step from c by means of t. All the transitions in the model are then applied iteratively on c to obtain the set of new candidate states c'. Hence, the more transitions are in the model, the more possible executions are generated by applying the transitions on the current configuration and the more time and memory are needed to compute the pre-images. Various are the sources of the growth of the number of transitions but mainly the use of flags, the nondeterministic updates and the transitions of the automata in Figure 2 play the major role. Limiting the size of the model was achieved also through the following simplification: the emit state of a bolt is enforced if a bolt is ready to emit, through transition (1); state *take* was omitted and its associated transition was removed. Finally, to reduce the number of flag transitions, we chose to restrict the reachability analysis only to one bolt of the system otherwise the analysis could not terminate even in trivial cases.

5. CONCLUSIONS AND RESULTS

We initially ran experiments with both MCMT and Cubicle, but we finally settled on the latter since it supports bidimensional arrays, which allowed us to write simpler and more concise models. We ran the verification of the model[4] obtained after many refinements to overcome limitations and fixing flaws. The analysis aimed to check the safety of the queue length of the bolt with index 1 for two different values of Ts_{min}. The limit on the queue was set initially to 3 (tuples). We chose this value because, in the case of an unsafe system, it should have allowed the construction of a reasonably short execution leading the system to an unsafe state. In fact, given that the time to process a tuple is 1 time unit, 4 consecutive spout emits, interleaved with time elapse transitions, would lead the system to the unsafe state $L[1] \geq 3$, if $Ts_{min} < 1$; whereas the system is safe for $Ts_{min} > 1$. Our experimental results show that the current version of the model is still not tractable and verification of such a topology model is not doable at the moment. Actually, both the verification instances could not terminate the computation because of memory exhaustion.

Hence, we set the limit of the queue to 2. The result, obtained in less than 1 minute, was that the system is unsafe for $Ts_{min} < 1$. The exhibited error trace is: $Init \rightarrow time_elapse \rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow$
$\rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow time_elapse \rightarrow setDoTake_{\texttt{True}} \rightarrow$
$\rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{bolt1}_{\textbf{take}} \rightarrow setDoTake_{\texttt{False}} \rightarrow$
$\rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout_emit} \rightarrow time_elapse \rightarrow$
$\rightarrow setDoTake_{\texttt{False}} \rightarrow setDoEmit_{\texttt{False}} \rightarrow \textbf{spout}_{\textbf{emit}} \rightarrow$
$\rightarrow L[1] \geq 2$. A detailed analysis of the outcome shows that the two spout emits after the take action performed by B_1 fill its queue with two tuples.

For $Ts_{min} > 1$, we expect that the system is safe, however the verification did not terminate.

The results obtained from our modeling exercises show that it is worth verifying safety properties of Storm applications which generalize over the number of processes. However, they also show that better abstractions for Storm topologies are needed, ones that are tractable with the tools at the state of the art, but also expressive enough to model realistic applications.

Acknowledgements. Work supported by Horizon 2020 project no. 644869 (DICE). We thank Alain Mebsout for support with the Cubicle tool.

6. REFERENCES

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

[2] M. M. Bersani, M. Rossi, and P. San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. In *Proc. of TIME*, pages 99–106, 2013.

[3] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. D. Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. Perez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladušič. DICE: Quality-driven development of data-intensive cloud applications. In *Proc. of MiSE*, pages 78–83, 2015.

[4] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT model checking of array-based systems. In *Proc. of IJCAR*, pages 67–82, 2008.

[5] F. Marconi, M. M. Bersani, M. Erascu, and M. Rossi. Towards the formal verification of data-intensive applications through metric temporal logic. In *Proc. of ICFEM*, pages 193–209, 2016.

[4]https://github.com/merascu/DICE-StormModellingFOL

7. APPENDIX

For a better understanding of our paper, and due to the space constraints, we present here more details on the final refinement of the model only for supporting the review process of the work.

As mentioned in Section 4, we faced some challenges, for example the *incorrect firing of transitions* and the *large number of transitions*.

The first one required the introduction two boolean flags, *DoEmit* and *DoTake*, and one ternary variable *SetFlags*, with values in $\{\texttt{setDoTake}, \texttt{setDoEmit}, \texttt{sysTr}\}$, to realize the effect of urgent transitions. The value of *SetFlags* enables, circularly and with a strict order, the system transitions, for the actions *emit* for spouts and bolts and *take* for bolts, and flag transitions, those defining the values of *DoEmit* and *DoTake* only. In the model, the system transitions, that is $spout_{emit}$, $bolt_{emit}$ and $bolt_{take}$, can fire only if the associated flag, that is *DoEmit* - for bolt/spout emit, *DoTake* - for bolt take, is `true` and when *SetFlags* allows the firing of system transitions. For each flag, we used various flag transitions to set its value.

Avoiding a large number of transitions, in addition to those needed to realize nondeterministic updates and those ones which implementing the model that were detailed in Section 4, can be achieved through a suitable configuration of the *granularity of the flags*. By granularity of flags we mean how many bolts are took into account in defining the transitions that set the negative value of a flag. Actually, this number affects how many of transitions are needed to update a flag with `false`, as for each possible condition leading a flag to `false`, a suitable transition with its specific guard has to be included in the model. However, only one transition is enough to set a flag to `true`. For instance, to update *DoTake* with `true` it is enough to check if there exist a bolt with non empty queue and a process which is in *idle* state. However, the negative value requires a case analysis that involves all the bolts and various situations. The system cannot perform a bolt take if either all the bolts have empty queue or, for each of them, the queue is non-empty but no process is in idle state. To reduce the number of transitions defining the flag values, we chose to restrict the reachability analysis only to one bolt of the system otherwise the analysis could not be finished due to memory limit. This allowed to include 8 flag transitions for both *DoTake* and *DoEmit*. In the general case, if we consider the topology in Figure 2, the number of flag transitions is greater than 20 and bigger than the number of system transitions.

7.1 Initial state

The initial state (4) essentially expresses that the time clock is set to 0, all spouts and bolts are in the idle state, the level of occupancy of the queues associated to the bolts and the number of tuples processed are 0, the emission time of the spouts is 0, number of tuples that are currently processed is 0 and the number of active processes in any bolt i is strictly positive and not null.

$$T = 0.0 \land CanTimeElapse = \texttt{true} \land \texttt{DoTake} = \texttt{false} \land \texttt{DoEmit} = \texttt{false} \land SetFlags = \texttt{No} \land \tag{4}$$

$$\forall_{i,x} \; (B[i,x] = \texttt{I} \land L[i] = 0.0 \land P[i,x] = 0.0 \land Stime[i] = 0.0 \land 0 < N_{proc}[i]) \tag{5}$$

7.2 Flag Transitions

We outline the semantics only of the flag transitions that change the value of flag *DoTake*, as the arguments for *DoEmit* are similar.

Transition $setDoTake_T$ (6) sets the value of *DoTake* to true when Bolt 1 has an active process x which is idle and its queue $L[i]$, with $i = 1$, is non empty.

$$setDoTake_T : \quad \exists_{i,x} \; SetFlags = \texttt{DT} \land i = 1 \land 0 \leq x < N_{proc}[i] \land L[i] > 0.0 \land B[i,x] = I \land DoTake' = true \land SetFlags' = \texttt{DE} \tag{6}$$

Variable *DoTake* is updated with false in the following two cases:

1. the queue of Bolt 1 is empty, i.e., $L[i] = 0$ and $i = 1$ (transition $setDoTake_{F1}$ (7)).

2. The queue of Bolt 1 in non empty but all the active processes are not idle (transitions $setDoTake_{F2}$ (8) and $setDoTake_{F2a}$ (9)).

To express the second condition in the implemented model we need two different transitions with different guards. The reason behind this construction is that the tool allows us to universally quantify a variable x on a domain which does not contain the values that are assigned to the existentially quantified variables of the formula transition. Therefore, we have to distinguish the case $N_{proc}[i] > 1$, which is handled by $setDoTake_{F2}$, from the case $N_{proc}[i] = 1$ of transition $setDoTake_{F2a}$. In the former case, $\forall_{y \neq j}$ enforces that $B[j,y]$ is not idle for all the y that are not equal to j, from 0 to N_{proc} excluded. In the latter one, when $N_{proc}[i] = 1$, the universal quantification is vacuously true and the condition (2) is guaranteed by $B[j,x] \neq \texttt{I} \land j = 1 \land x = 0$. In all the previous transitions, variable *SetFlag* is set to `DE` in order to enforce the evaluation of flag *DoEmit* in the next transition. After performing a transition for setting *DoEmit*, flag *setFlags* is updated with `No` to allow the firing of system transitions.

$$setDoTake_{F1} : \quad SetFlags = \texttt{DT} \land j = 1 \land L[j] = 0 \land \texttt{DoTake}' = \texttt{false} \land SetFlags' = \texttt{DE} \tag{7}$$

$$\begin{aligned} setDoTake_{F2} : \quad & \exists_{j} \; SetFlags = \texttt{DT} \land j = 1 \land L[j] > 0.0 \land j < N_{proc}[j] \land B[j,j] \neq \texttt{I} \land \\ & \forall_{y \neq j} (y < 0 \lor y \geq N_{proc}[j] \lor B[j,y] \neq \texttt{I}) \land \; \texttt{DoTake}' = \texttt{false} \land SetFlags' = \texttt{DE} \end{aligned} \tag{8}$$

$$\begin{aligned} setDoTake_{F2a} \quad & \exists_{j,x} \; SetFlags = \texttt{DT} \land j = 1 \land L[j] > 0.0 \land j \geq N_{proc}[j] \land B[j,x] \neq \texttt{I} \land x = 0 \land \\ & \forall_{y \neq j} (y < 0 \lor y \geq N_{proc}[j] \lor B[j,y] \neq \texttt{I}) \land \; DoTake' = \texttt{false} \land SetFlags' = \texttt{DE} \end{aligned} \tag{9}$$

The set of transitions for setting the value of *DoEmit* to true, respectively false, is as follows.

$$\begin{array}{ll} \mathit{setDoEmit}_T: & \underset{i,x}{\exists}\ \mathit{SetFlags} = \mathtt{DE} \wedge i = 1 \wedge 0 \leq x < N_{proc}[i] \wedge B[i,x] = \mathtt{E} \wedge \\ & \mathit{DoEmit}' = \mathtt{true} \wedge \mathit{SetFlags}' = \mathtt{No} \end{array}$$

$$\begin{array}{ll} \mathit{setDoEmit}_{F1}: & \underset{j}{\exists}\ \mathit{SetFlags} = \mathtt{DE} \wedge j = 1 \wedge j < N_{proc}[j] \wedge B[j,j] \neq \mathtt{E} \wedge \\ & \underset{y \neq j}{\forall} (y < 0 \vee y \geq N_{proc}[j] \vee B[j,y] \neq \mathtt{E}) \wedge\ \ \mathit{DoEmit}' = \mathtt{false} \wedge \mathit{SetFlags}' = \mathtt{No} \end{array}$$

$$\begin{array}{ll} \mathit{setDoEmit}_{F1a} & \underset{j,x}{\exists}\ \mathit{SetFlags} = \mathtt{DE} \wedge j = 1 \wedge j \geq N_{proc}[j] \wedge B[j,x] \neq \mathtt{E} \wedge x = 0 \wedge \\ & \underset{y \neq j}{\forall} (y < 0 \vee y \geq N_{proc}[j] \vee B[j,y] \neq \mathtt{E}) \wedge\ \ \mathit{DoEmit}' = \mathtt{false} \wedge \mathit{SetFlags}' = \mathtt{No} \end{array}$$

7.3 System transitions

The set of core transitions is represented by: (1) three transitions describing the behavior of the Storm topology according to the possible states (see Figure 2) and how these influence the accumulation of tuples in the queues and (2) one transition which models the time elapsing in the system.

7.3.1 Spout emit

Spout emit in the topology is expressed by the transition $spout_{emit}$ (10).

$$\underset{i,j}{\exists}\ Ts_{min} < \mathit{Stime}[i] \wedge \mathit{SubscribedBS}[j,i] = \mathtt{true} \wedge \mathit{DoTake} = \mathtt{false} \wedge \mathit{DoEmit} = \mathtt{false}\ \wedge\ \mathit{SetFlags} = \mathtt{No}\ \wedge$$
$$\underset{l}{\forall} \left(\begin{array}{lll} L'[l] = & \textbf{if}\ (l = j)\ \textbf{then}\ L[l] + 1\ \textbf{else}\ L[l] & \wedge \\ \mathit{Stime}'[l] & \textbf{if}\ (l = i)\ \textbf{then}\ 0\ \textbf{else}\ \mathit{Stime}[l] & \wedge \\ \mathit{CanTimeElapse}' = & \mathtt{true} & \wedge \\ \mathit{SetFlags}' = & \mathtt{DT} & \end{array} \right) \tag{10}$$

The transition modifies the length of the queues of the bolts connected with the emitting spouts and prepares the emitting spout for a new emit (by resetting to 0 its clock variable *Stime*). Spout emit is activated if the minimum time between two consecutive spout emit performed by the same bolt (*Tsmin*) is smaller than the time since last spout emit (*Stime*), if there exists a bolt j subscribed to spout i ($\mathit{SubscribedBS}[j,i] = \mathtt{true}$), and if the spout is enable to emit. This last condition is guaranteed when *DoTake* is false, *DoEmit* is false (meaning that Bolt 1 is not enabled for taking and emitting a tuple) and *SetFlags* is false, i.e., flag transitions are disabled. In these conditions:

1. the length of the queue of the bolt j is increased by 1;
2. emission time of the spout i is reset;
3. time elapsing transition (13) can fire by setting $\mathit{CanTimeElapse}'$ to `true` and
4. flag *SetFlags* is set to `DT` to enable flag transitions and guarantee the alternation among system and flag transitions. $\mathit{SetFlags} = \mathtt{DT}$ enforces first the evaluation of flag *DoTake* through the next transition.

7.3.2 Bolt emit

Bolt emit is expressed by the transition $bolt_{emit}$ (11).

$$\underset{i,j,x}{\exists}\ B[i,x] = \mathtt{E} \wedge \mathit{SubscribedBB}[j,i] = \mathtt{true} \wedge 0 \leq x < N_{proc}[i] \wedge \mathit{SetFlags} = \mathtt{No} \wedge ((i = 1 \wedge \mathtt{DoEmit} = \mathtt{true}) \vee i \neq 1))$$
$$\underset{l,z}{\forall} \left(\begin{array}{lll} L'[l] = & \textbf{if}\ (l = j)\ \textbf{then}\ L[l] + 1\ \textbf{else}\ L[l] & \wedge \\ B'[l,z] = & \textbf{if}\ (z = x \wedge l = i)\ \textbf{then}\ \mathtt{I}\ \textbf{else}\ B[l,z] & \wedge \\ \mathit{CanTimeElapse}' = & \mathtt{true} & \wedge \\ \mathit{SetFlags}' = & \mathtt{DT} & \end{array} \right) \tag{11}$$

The transition modifies the length of the queues of the bolts connected with the emitting bolts while the emitting bolts become idle. The transition is activated if there exists a bolt i which is in the emit state, if there is another bolt j which is subscribed to it and if flag transitions are disabled $\mathit{SetFlags} = \mathtt{No}$ and either the emitting bolt is 0 or 2 (then $i \neq 1$) or bolt 1 is allowed for emitting, i.e., $i = 1 \wedge \mathit{DoEmit} = \mathtt{true}$. If this is the case, then:

1. the length of the queue of the bolt j is increased by 1;
2. the emitting bolt becomes idle;
3. time elapsing transition (13) is enabled by setting *CanTimeElapse* to `true` and
4. flag *SetFlags* is set to `DT` to enable flag transitions.

7.3.3 Bolt take

Bolt take is expressed by the transition $bolt_{take}$ (12).

$$\underset{j,y}{\exists}\ B[j,y] = \mathtt{I} \wedge L[j] \geq 1.0 \wedge 0 \leq y < N_{proc}[j] \wedge \mathit{SetFlags} = \mathtt{No} \wedge ((j = 1 \wedge \mathit{DoTake} = \mathtt{true}) \vee j \neq 1)$$
$$\underset{l,z}{\forall} \left(\begin{array}{lll} L'[l] = & \textbf{if}\ (l = j)\ \textbf{then}\ L[l] - 1\ \textbf{else}\ L[l] & \wedge \\ B'[l,z] = & \textbf{if}\ (z = y \wedge l = j)\ \textbf{then}\ \mathtt{X}\ \textbf{else}\ B[l,z] & \wedge \\ P'[l,z] = & \textbf{if}\ (z = y \wedge l = j)\ \textbf{then}\ 1\ \textbf{else}\ P[l,z] & \wedge \\ \mathit{CanTimeElapse}' = & \mathtt{true} & \wedge \\ \mathit{SetFlags}' = & \mathtt{DT} & \end{array} \right) \tag{12}$$

The transition decrements of one unit the queue of the bolt which performs a take action and activate an idle thread by changing its state to X and by setting the remaining amount $P[j, y]$ of that process to 1. The transition is enabled when there is a bolt j and a process y such that $B[i, y]$ is idle, the queue of bolt j is non empty ($L[j] \geq 1.0$), the process y is an instantiated valid process of j as it is between 0 and N_{proc} and, finally, either the taking bolt is 0 or 2 (then $i \neq 1$) or bolt 1 is allowed for taking, i.e., $i = 1 \wedge DoTake = \texttt{true}$. If this is the case, then:

1. the length of the queue of the bolt j is decreased by 1;
2. the taking bolt becomes ready to execute;
3. the amount of processing that process y has to perform is set to 1;
4. time elapsing transition (13) is enabled by setting $CanTimeElapse$ to `true` and
5. flag $SetFlags$ is set to `DT` to enable flag transitions.

7.3.4 Time elapse

Time elapsing in the topology is expressed by the transition $time_{elapse}$ (13).

$$0 < C \ \wedge \ CanTimeElapse = \texttt{true} \wedge \\ \underset{j,z}{\forall} \left(\begin{array}{ll} T' = & T + C \\ P'[j,z] = & \textbf{if } (0 \leq P[j,z] - c) \textbf{ then } P[j,z] - c \textbf{ else } 0 \wedge \\ B'[j,z] = & \textbf{if } (B[j,z] = \texttt{X} \wedge j = 0 \wedge 0 \leq z < N_{proc}[j] \wedge P[j,z] \leq c) \textbf{ then } \texttt{E} \\ & \textbf{elseif } (B[j,z] = \texttt{X} \wedge j = 1 \wedge 0 \leq z < N_{proc}[j] \wedge P[j,z] \leq c) \textbf{ then } \texttt{E} \\ & \textbf{elseif } (B[j,z] = \texttt{X} \wedge j = 2 \wedge 0 \leq z < N_{proc}[j] \wedge P[j,z] \leq c) \textbf{ then } \texttt{I} \\ & \textbf{else } B[j,z] \wedge \\ Stime'[j] & Stime[j] + C \wedge \\ CanTimeElapse' = & \texttt{false} \wedge \\ SetFlags' = & \texttt{DT} \end{array} \right) \tag{13}$$

Formula (13) states that if there exists a positive value C and $canTimeElapse = \texttt{true}$ then:

1. the time progresses by incrementing T with C time units;
2. for all the j and z, if C is not big enough to complete the remaining part of the computation (when $P[j, z] - C > 0$) then $P[j, z]$ is updated with the progress $P[j, z] - C$; otherwise, if C allows the bolt to complete the current processing (when $P[j, z] - C < 0$) then $P[j, z]$ is set to 0;
3. if bolts with the index 0 or 1 complete the processing then they can enter into the emit state; if Bolt 2 is executing then it go into idle state (as Bolt 2 is a final component that is not connected to any other element of the topology); otherwise they do not change the state;
4. the emit time for all spouts increases by C;
5. $CanTimeElapse'$ becomes `false` which means that the system must fire next one of the transitions $spout_{emit}$, $bolt_{emit}$, $bolt_{take}$;
6. the flag transitions must be fired next since $SetFlags$ is set to `DT`.