CSTN Benchmarks

Roberto Posenato
Dipartimento di Informatica
Università degli Studi di Verona, Italy

April 8, 2024

1 Introduction

During the research about temporal networks, we prepared some benchmarks for testing some algorithms:

- CSTNBenchmark2016 and CSTNBenchmark2018: for testing Dynamic Consistency (DC) checking algorithms for Conditional Simple Temporal Networks (CSTNs).
- CSTNUBenchmark2018: for testing Dynamic Controllability (DC) checking algorithms for Conditional Simple Temporal Networks with Uncertainty (CSTNUs).
- STNUBenchmark2020: for testing DC checking algorithms for Simple Temporal Networks with Uncertainty (STNUs).
- CSTNPSUBenchmarks2023: for testing DC checking algorithm and Prototypal Link algorithm for Conditional Simple Temporal Networks with Partially Shrinkable Uncertainty (CSTNPSUs) (a.k.a. FTNUs).
- OSTNUBenchmarks2024: for testing Agile Controllability algorithm for Simple Temporal Networks with Uncertainty and Oracles (OSTNUs).

CSTNBenchmark2016 and CSTNBenchmark2018 benchmarks contain CSTN instances, CSTNUBenchmark2018 contains CSTNU instances, STNUBenchmark2020 contains STNU instances, CSTNPSUBenchmarks2023 contains CSTNPSU instances, and OSTNUBenchmarks2024 constains OSTNU instances.

All CSTN/CSTNU/CSTNPSU instances were determined transforming random workflows generated by the ATAPI Toolset [1], while STNU ones were determined using ad-hoc random generator present in the CSTN Tool library.

The original version of this document can be download at `http://profs.scienze.univr.it/~posenato/software/cstnu/tex/benchmarks.pdf`

1.1 Temporal Networks From Random Workflows

We considered a random workflow generator as source of random CSTN((PS)U) instances for having a closer approximation to real-world instances and also for generating networks that are more difficult to check than those created haphazardly.

The ATAPI toolset produces random workflows according to different input parameters that govern the number of tasks, the probability of having parallel (AND)/alternative (XOR) branches, the probability of inter-task temporal constraints, and so on. In particular, the ATAPIS toolset builds a workflow instance in two phases. In the first phase, it prepares the structure of the network in a recursive way starting from a pool of N blocks representing tasks. At each cycle, randomly it removes two blocks from the pool and casually decides how to combine them using an AND/XOR/SEQUENTIAL connector. The resulting block is then added to the pool. The generation phase ends when there is only one block in the pool. The probability to choose an AND or an XOR connector can be given as input (P_A, and P_C parameters). Temporal constraints are added in the second phase considering some parameters that can be given as input. The following parameter-values were determined after some experiments where we tried to discover a good mix of them to guarantee not trivial consistent/not consistent instances. The chosen values were:

- each task duration is a subrange of $[2, 30]$; each bound is chosen according to a beta distribution;
- each connector duration has always range $[1, 10]$;
- each delay between sequential tasks is a subrange of $[1, 25]$; each bound is according to a beta distribution;
- each delay between a connector and a task is fixed to $[1, 100]$;
- the probability to have an inter-activity temporal constraint between any pair of activities (i.e, tasks or connectors) was set to 0.2. In case that an inter-activity temporal constraint is set, its range is a casual subrange of the $[mindistance, maxdistance]$ between the two activities, where min and max distances were calculated by the tool.

For generating CSTN instances, the method consisted in two phases:

- Fixed the above parameters, workflow graphs were randomly generated by varying the number of tasks (N), the probability for parallel branches (P_A), and the probability for alternative branches (P_C) to determine different blocks of instances;

- each workflow graph was then translated into an equivalent CSTN using the method proposed in [2]. Task durations were represented as ordinary constraints in CSTN instances.

For generating the CSTNU instances, the method used was similar to the above one for CSTN benchmarks. The only difference was that all task durations were translated as contingent links in corresponding CSTNU instances.

For generating the CSTNPSU instances, we considered the CSTNU ones and we transformed each contingent link into a guarded one adding the two external bounds to the contingent core. After some trials, we found that it was sufficient to enlarge the contingent core of about 10% for having a suitable guarded range.

1.2 Temporal Networks From a Random Generator

During the study of DC checking algorithm for STNU, we wanted to test the different polynomial-time algorithms using instances having significant size, i.e., a size much larger than the size of instances obtained in previous benchmarks for CSTN or CSTNU. We verified that increasing the size of random workflow instances built using ATAPI toolset to have bigger STNU instances did not work because, for the characteristics of generated workflows, the obtained instances were almost not DC and the generation of few DC instances required a lot of computation.

Therefore, we decided to build a specific STNU random generator (it.univr.di.cstnu.util.STNURandomGenerator) capable of building big-size STNU instances. Our STNU generator can build random instances having a chosen topology that can be tuned by a variety of input parameters. The possible topologies are *1) no-topology, 2) tree*, or *3) worker-lanes*. After some testing, we verified that the *worker-lanes* topology, which simulates the swimming-pools (with one lane each) of business process modeling [3], is the most interesting because it allows the generation of big random instances where there could be circuits involving many constraints. In this topology, the set of contingent links is partitioned into a given number of lanes. Contingent links within each lane are interspersed with ordinary constraints that specify delays between the end of one contingent link and the start of the next. Finally, there are constraints between pairs of nodes belonging to different lanes to represent temporal-coordination constraints among time-points of different swimming-pools. Typically, such constraints involve nodes on different lanes that are at a similar distance from the start of their respective lanes.

As an example, Figure 1 depicts a portion of a random STNU having 500 nodes and 50 contingent links in a 5-worker-lane topology.

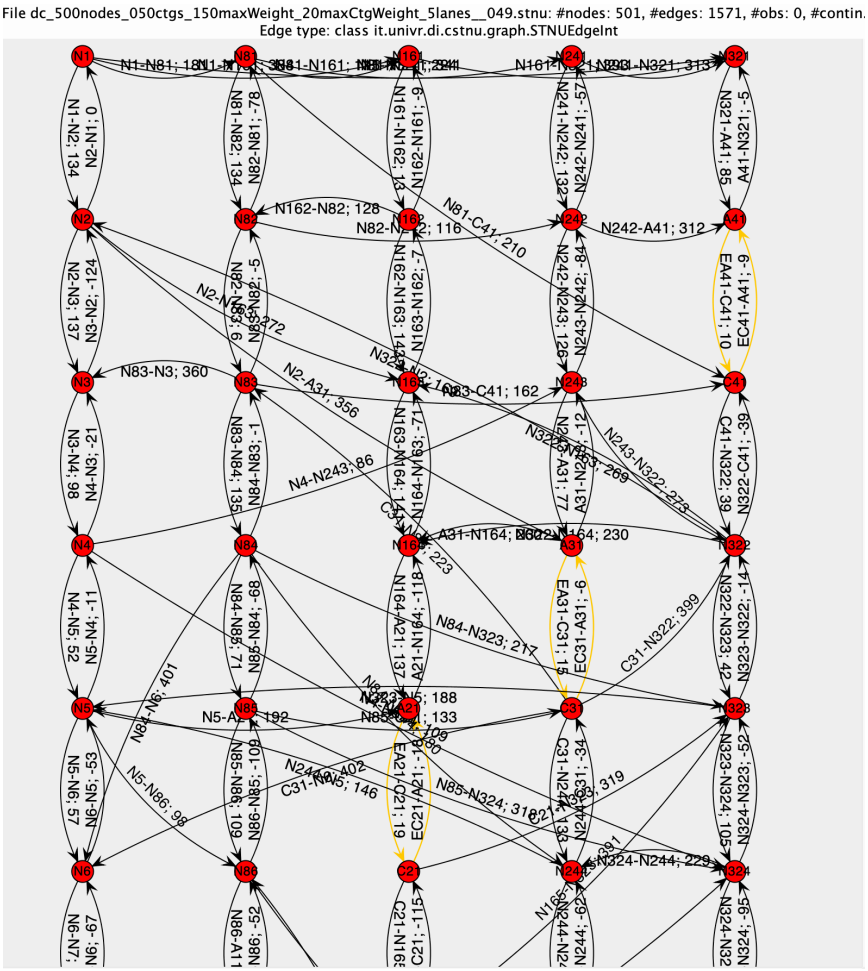


Figure 1: An example of a randomly generated STNU

Many aspects of the worker-lanes topology can be tuned as input parameters; the number of nodes, the number of contingent links, the number of lanes, the probability of a temporal constraint for a pair of nodes from different lanes, the maximum weight of each contingent link, the maximum weight of each ordinary constraint, and so on.

2 Benchmarks

2.1 CSTNBenchmark2016

This benchmark is composed of four sub-benchmarks (named as *benchmarks* in papers [4, 5, 6, 7, 8, 9]). Each sub-benchmark is composed of two sets: one set of DC CSTN instances and one set of NOT DC ones.

Each sub-benchmark, characterized by a number N, is called **Size**N and contains CSTN instances (DC/NOT DC) generated by random workflows

having N tasks, k XOR connectors (= propositions in CSTN) and a variable number of AND connectors. The probability for parallel branches P_A was fixed to 0.2 as well as the probability for alternative branches P_C.

The following table summarizes the main characteristics of all sub-benchmarks:

benchmark:	**size10**	**size20**	**size30**	**size40**
#tasks:	10	20	30	40
k=#XOR:	3	5	7	9
#CSTN-nodes:	45-59	79-95	123-135	159-175

For each sub-benchmark, there are at least 60 dynamically consistent CSTNs and 20 non-dynamically consistent CSTNs. Since the original ATAPIS toolset allows one to fix only the probability of AND/XOR connectors, it was necessary to run the toolset a huge number of times for obtaining the instances with the above characteristics.

Different workflow graphs with the same number of tasks may translate into CSTNs of different sizes due to different numbers of AND connectors in the workflows. This fact represented the main weakness of this benchmark. For example, even if for $N = 10$ and $k = 3$ there are 60 DC instances, such instances are CSTN instances with different order (=#nodes). Few of them have the same order and this fact represents a limitation when an evaluation of DC execution time with respect of the CSTN order is required.

For this benchmark we do not report here the results obtained using our algorithms because they are superseded by the results obtained with CSTNBenchmark2018, presented in the next section.

2.2 CSTNBenchmark2018

The structure of this benchmark is equals to the CSTNBenchmark2016 benchmark one: four sub-benchmarks with two sets for each. The main differences are the number of instances and the structure of the instances.

After an important modification of ATAPI Toolset source code, it was possible to give as input also the number of XOR/AND connectors that a random workflow instance must have. Therefore, it was possible to build sets of more uniform instances where the randomness could decide how to mix components and constraints but not their quantities. It is possible to show that the relation between workflow component quantities and CSTN order is $5 + 2N + 6k + 4j$, where N is the number of tasks, k the number of XOR connectors, and j the number of AND connectors. Therefore, for each possible planned combinatios of #task-#XOR-#AND, it was possible to generate randomly 50 DC and 50 NOT DC instances. The following table summarizes the characteristics of each sub-benchmark.

Group	Benchmark	instance indexes	# activities	#XOR	#AND	CSTN order
Size010-3	B10-3-0	000-049	10	3	0	43
	B10-3-1	050-099	10	3	1	47
	B10-3-2	100-149	10	3	2	51
	B10-3-3	150-199	10	3	3	55
	B10-3-4	200-249	10	3	4	59
Size020-5	B20-5-0	000-029	20	5	0	75
	B20-5-1	030-059	20	5	1	79
	B20-5-2	060-099	20	5	2	83
	B20-5-3	090-119	20	5	3	87
	B20-5-4	120-149	20	5	4	91
Size030-7	B30-7-0	000-029	30	7	0	107
	B30-7-1	030-059	30	7	1	111
	B30-7-2	060-089	30	7	2	115
	B30-7-3	090-119	30	7	3	119
	B30-7-4	120-149	30	7	4	123
Size040-9	B40-9-0	000-029	40	9	0	139
	B40-9-1	030-059	40	9	1	143
	B40-9-2	060-089	40	9	2	147
	B40-9-3	090-119	40	9	3	151
	B40-9-4	120-149	40	9	4	155

The total number of CSTN instances is 2000, 1000 DC and 1000 not DC, divided in 4 main groups, in turn divided in other 4 groups having 50 DC and 50 NOT DC instances.

2.2.1 Experimental Evaluation

We used CSTNBenchmark2018 for testing 10 different DC checking algorithms for CSTN:

1. **Std**: it checks the DC property considering the standard semantics [5].
2. **Std-woNL**: it checks the DC property considering the standard semantics on the equivalent CSTN where there is no node labels [6].
3. **ϵ**: it checks the DC property considering the ϵ semantics [5].
4. **ϵ-woNL**: it checks the DC property considering the ϵ semantics on the equivalent CSTN where there is no node labels [6].
5. **ϵ-3R**: it checks the DC property considering the ϵ semantics and using only rules LP, $\text{qR}_{\mathbf{0}}$, and $\text{qR}^{*}_{\mathbf{3}}$.
6. **ϵ-3R-woNL**: it checks the DC property considering the ϵ semantics on equivalent CSTN where there is no node labels and using only rules LP, $\text{qR}_{\mathbf{0}}$, and $\text{qR}^{*}_{\mathbf{3}}$.

7. **IR**: it checks the DC property considering the instantaneous reaction semantics [5].

8. **IR-woNL**: it checks the DC property considering the instantaneous reaction semantics on the equivalent CSTN where there is no node labels [6].

9. **IR-3R**: it checks the DC property considering the instantaneous reaction semantics and using only rules LP, $\mathrm{qR}_{\mathbf{0}}$, and $\mathrm{qR}^{*}_{\mathbf{3}}$.

10. **IR-3R-woNL**: it checks the DC property considering the instantaneous reaction semantics on the equivalent CSTN where there is no node labels and using only rules LP, $\mathrm{qR}_{\mathbf{0}}$, and $\mathrm{qR}^{*}_{\mathbf{3}}$.

11. **π-3R-woNL**: it checks the DC property considering the fixed instantaneous reaction semantics (π) on the equivalent CSTN where there is no node labels [7].

12. **Potential**: it checks the DC property considering the fixed instantaneous reaction semantics (π) and using the potential function [9].

In the following we propose two diagrams that show the comparison of such algorithms using the CSTNBenchmark2018 benchmark.

We executed the algorithm implementations present in CSTNU Tool library using an Oracle JVM 8 in a Linux machine with an AMD Opteron 4334 CPU (12 cores) and 64GB of RAM.

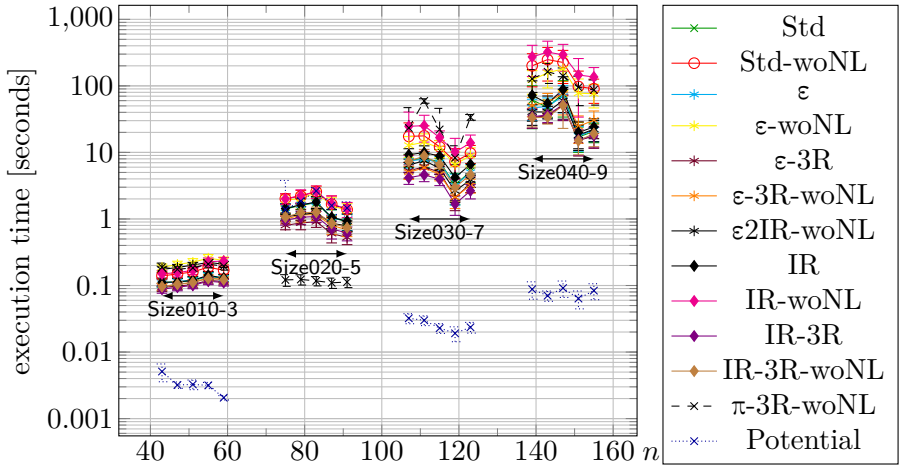
The execution times were collected by a Java program (`Checker`, present in CSTNU Tool) that allows one to determine the average execution time—and its standard deviation—of one DC checking algorithm applied to a set of CSTN instances. Moreover, `Checker` allows one to require to the operating system to allocate one or more CPU cores for executing the algorithm in sequential/parallel way on files without the rescheduling of threads in different cores during the execution. We experienced that even if AMD Opteron 4334 has 12 cores, the best performances were obtained only when all checks are made by only **one** core. We verified that the memory-accesses by cores represent a bottleneck that limits the overall performance. Therefore, all the data presented in this section were obtained using only one core (parameter `-nCPUs 1`).

The parameters for the Oracle Java Virtual Machine 1.8.0_144 were: `-d64`, `-Xmx6g`, `-Xms6g`, `-XX:NewSize=3g`, `-XX:MaxNewSize=3g`, `-XX:+UseG1GC`, `-Xnoclassgc`, and `-XX:+AggressiveOpts`.

The first diagram shows the average execution times of all algorithms with respect to the order of dynamic consistent CSTN instances. Each drawn value is the sample average $\bar{X}_{50}$ of execution times obtained considering the fifty instances of the relative benchmark. In details, $\bar{X}_{50} = \frac{\sum_{i=1}^{50} X_i}{50}$ where X_i is the average execution time obtained executing 3 times the algorithm on

instance having index i[1]. The error bar of each drawn value represents 2.010 times the standard error of the mean, $\frac{S_{50}}{\sqrt{50}}$, where S_{50} is the corrected sample standard deviation, $S_{50} = \sqrt{\frac{\sum_{i=1}^{50}(X_i - \bar{X}_{50})^2}{49}}$. Value 2.010 is the Student's t distribution value with 49 degrees of freedom. Therefore, the error bar represents a 95% confidence interval for the average execution time of the algorithm on instances having the main characteristics of the considered benchmark.

Figure 2: Consistent Instances



Although the diagram is quite crowded, it is possible to see (and data confirm) that the DC checking algorithm has the worst performance when it has to apply IR semantics without node labels while the fastest DC checking can be done using the **Potential** algorithm. The outstanding performance of **Potential** algorithm can be justified observing that this algorithm does not add constraints to the network but only potential values to nodes and that the quantity of such values is, in general, lower than the number of new constraints added by other algorithms.

We noted that the experimental data contain outliers, instances for which the execution time is quite far from the average execution time. Outliers represent hard instances for the DC checking problem (the problem was shown to be PSPACE-complete).

The following figures show the distribution of execution time of **IR-3R** in terms of quartiles in the groups Size030-7 and Size040-9. Each box has the lower edge equal to the first quartile (Q_1) while the upper edge equal to the third one (Q_3). The edge inside each box represents the median of the sample. Horizontal edges outside a box represent the *whiskers*. The lower whisker value is the smallest data value which is larger than $Q_1 - 1.5 \cdot \text{IQR}$,

[1]The determination of all values required to execute the DC checking for 24 000 times for a total of 83.18 hours.

where IQR is the *inter–quartile–range*, i.e., $Q_3 - Q_1$. The upper whisker is the largest data value which is smaller than $Q_3 + 1.5 \cdot \text{IQR}$. Diamonds above the upper whisker represent the data value outliers. The diamond in the highest position in each set of data represents the worst case value of the benchmark. Diamond inside a box represents the average value of the benchmark.

Figure 3: Execution time distribution of IR-3R in Size030 group

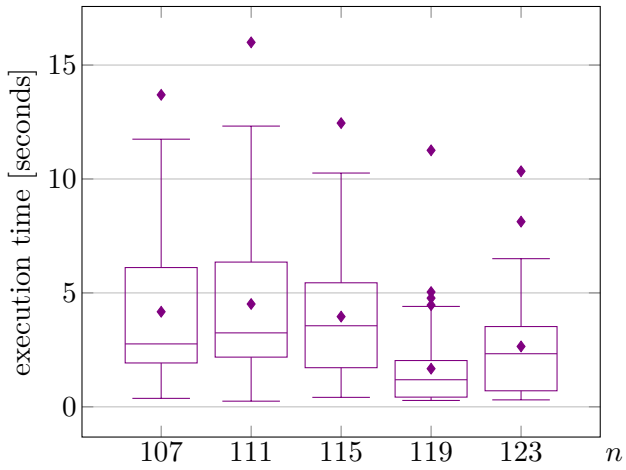
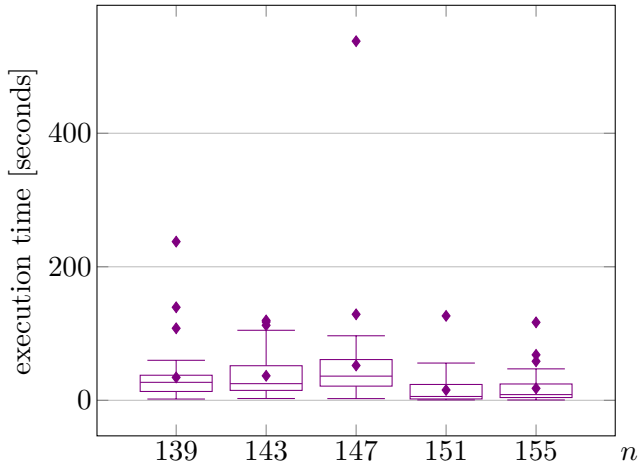


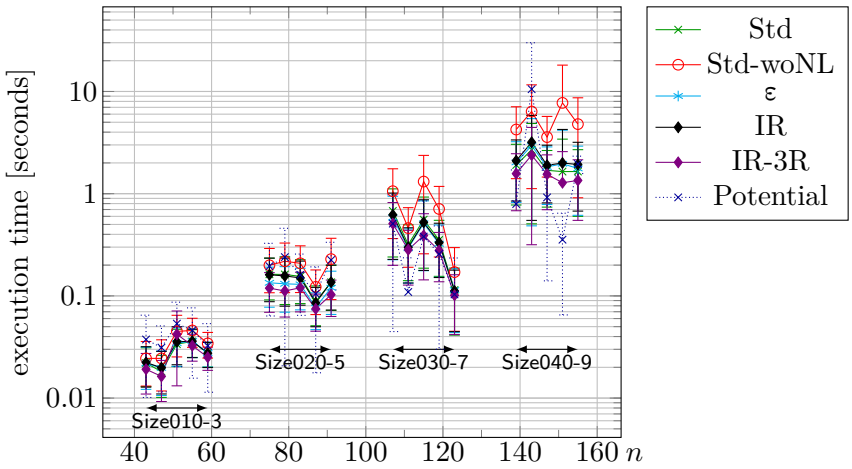
Figure 4: Execution time distribution of IR-3R in Size040 group



The previous two diagrams show clearly that there are few instances that bias the value of sample average in a relevant way.

The following diagram shows the average execution times of some checking algorithms when CSTN instances are not consistent.

Figure 5: Not Consistent Instances



Again, even if the diagram is quite crowded, it is evident that Std-woNL shows the worst performance while IR-3R requires the minimum execution time for almost of the group of instances. Algorithm Potential has not an outstanding performance like for DC instances because the presence of negative circuits is, in general, determined promptly and, therefore, the execution time cannot be different significantly. The most important fact about these results is that, in general, checking NON DC instances requires less than an order of magnitude of execution time required for checking DC instances.

2.3 CSTNUBenchmark2018

The structure of this benchmark is similar to the CSTNBenchmark2018 one with two differences: the duration of each task is converted as contingent link and the number of instances is smaller.

Due to the internal building function of ATAPIS Toolset, the relation between workflow component quantities and CSTNU order is $5+2N+6k+6j$, where N is the number of tasks, k the number of XOR connectors, and j the number of AND connectors. The following table summarizes the characteristics of each sub-benchmark.

Group	Benchmark	instance indexes	#tasks	#XOR	#AND	CSTNU order
Size010-3	B10-3-0	000-049	10	3	0	43
	B10-3-1	050-099	10	3	1	49
	B10-3-2	100-149	10	3	2	55
	B10-3-3	150-199	10	3	3	61
	B10-3-4	200-249	10	3	4	67
Size010-4	B10-4-0	000-049	10	4	0	49
	B10-4-1	050-099	10	4	1	55
	B10-4-2	100-149	10	4	2	61
	B10-4-3	150-199	10	4	3	67
	B10-4-4	200-249	10	4	4	73
Size010-5	B10-5-0	000-049	10	5	0	55
	B10-5-1	050-099	10	5	1	62
	B10-5-2	100-149	10	5	2	67
	B10-5-3	150-199	10	5	3	73
	B10-5-4	200-249	10	5	4	79

The total number of CSTNU instances is 1500, 750 DC and 750 not DC, divided in 3 main groups—Size010-3, Size010-4, and Size010-5—in turn divided in other 5 groups having 50 DC instances each and in other 5 groups having 50 NOT DC instances each.

2.3.1 Experimental Evaluation

There are three implementations of CSTNU DC checking algorithm:

1. **STD**: it implements the CSTNU DC checking rules assuming instantaneous reaction in a streamlined CSTNU. The CSTNU DC checking rules are zqR0, zqR3, zlabeledLetterRemovalRule, labeledLetterRemovalRule, labeledPropagationqLP, and labeledCrossLowerCaseRule.

2. **STD OnlyToZ**: it is similar to STD version but it limits the propagation to edges heading to node Z. Therefore, it applies rules zqR0, zqR3, zlabeledLetterRemovalRule, zlabeledPropagationqLP, and zlabeledCrossLowerCaseRule.

3. **CSTNU2CSTN**: it determines the CSTNU DC status transforming the given CSTNU into an equivalent CSTN and checking the DC of this last one.

In the following we propose some diagrams that show the execution times of all versions in the CSTNUBenchmark2018 benchmark.

The implementations were developed in Java 8 and run on an Oracle JVM 8 in a Linux machine with an Intel(R) Xeon(R) CPU E5-2637 v4 3.50GHz and 503GB of RAM.

The execution times were collected by a Java program (`Checker`, proposed in our package) that allows one to determine the average execution time—and its standard deviation—of a DC checking algorithm applied to a set of instances.

The parameters for the Oracle Java Virtual Machine 1.8.0_144 were: `-d64`, `-Xmx6g`, `-Xms6g`, `-XX:NewSize=3g`, `-XX:MaxNewSize=3g`, `-XX:+UseG1GC`, `-Xnoclassgc`, and `-XX:+AggressiveOpts`.

All average execution times were determined considering only DC instances in benchmarks of groups Size010-3, Size010-4, and Size010-5. Each drawn value is the sample average $\bar{X}_{50}$ of execution times obtained considering fifty instances of the relative benchmark. In details, $\bar{X}_{50} = \frac{\sum_{i=1}^{50} X_i}{50}$ where X_i is the average execution time obtained executing 3 times the algorithm on instance having index i[2]. The error bar of each drawn value represents 2.010 times the standard error of the mean, $\frac{S_{50}}{\sqrt{50}}$, where S_{50} is the corrected sample standard deviation, $S_{50} = \sqrt{\frac{\sum_{i=1}^{50}(X_i - \bar{X}_{50})^2}{49}}$. Value 2.010 is the Student's t distribution value with 49 degrees of freedom. Therefore, the error bar represents a 95% confidence interval for the average execution time of the algorithm on instances of the considered benchmark.

In more details, Figure 6 shows the performances of the three implementations in benchmarks of group Size010-3. Each printed value corresponds to the average value determined considering the corresponding sub-benchmark B10-3-*. We verified some run time-out (45 minutes) for **STD** and **STD OnlyToZ** algorithm. The average execution time was determined considering also such time-outs and, therefore, it represents a lower bound to the real average.

All three implementations require, in average, a smaller execution time as the order of instances increases. This is due to the fact the bigger instances are obtained increasing the number of parallel connectors in the generated workflows (the number of contingent links and the number of observation node are fixed). Increasing the number of parallel connectors, there are more parallel branches and, therefore, more contingent links must be put in the same scenario. This makes a network easier to be checked.

From the experimental results, it emerges that **CSTNU2CSTN** has the best performance and the worst performance for all the implementations is in the sub benchmark B10-*-0 where all generated workflows have no parallel branches.

[2]The determination of all values required to execute the DC checking for 24 000 times, 83.18 hours.

Figure 6: Controllable Instances in group Size010

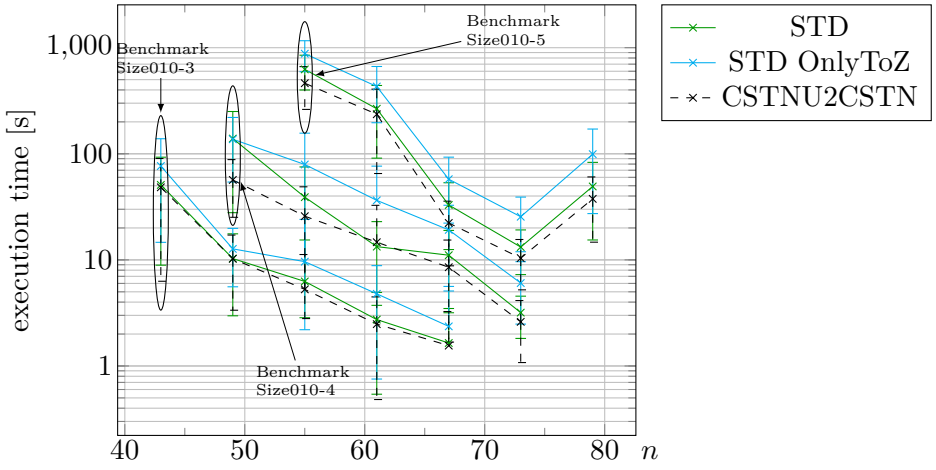
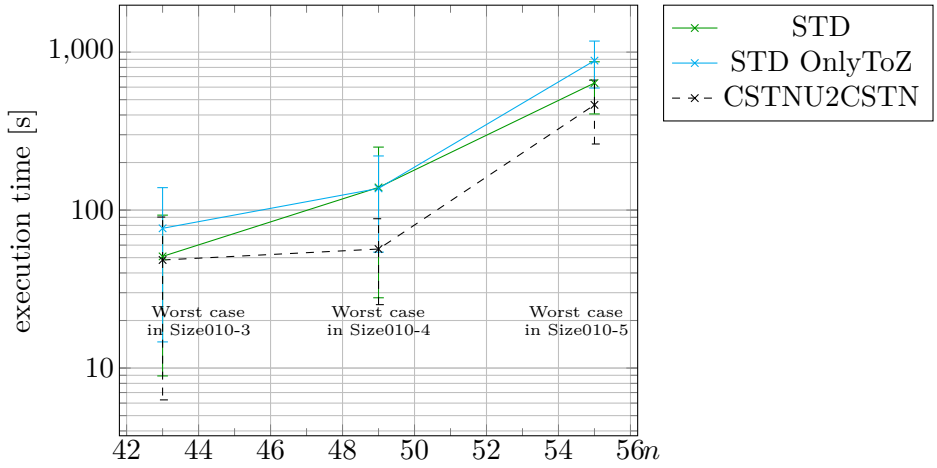


Figure 7 shows a detail about the worst-case execution time. For each sub-benchmark Size10-*-0, i.e., instances derived by workflows without parallel flows, we report the average execution-time to show how the average execution-time increases as the number of the choices increases.

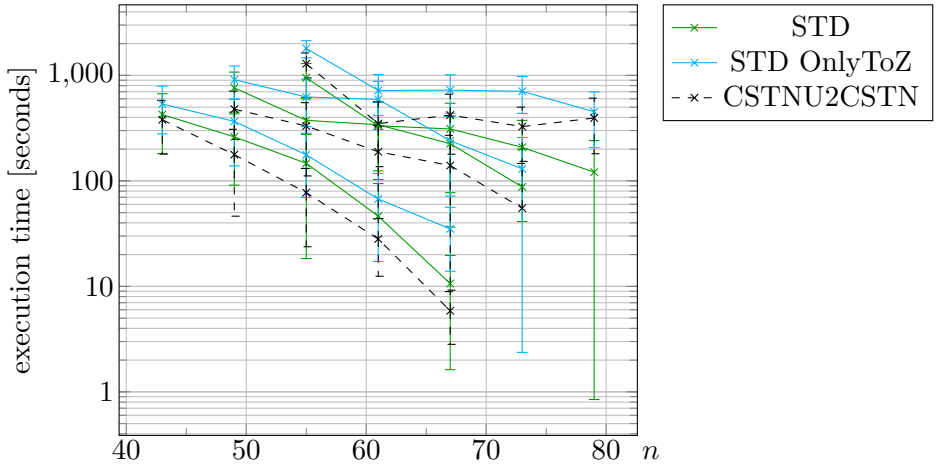
Figure 7: Worst Controllable Instances in group Size010



Again, it is clear that the **CSTNU2CSTN** has the best performance.

Figure 8 shows the average execution-time obtained when instances are not DC. The algorithms have a worse performance checking non-DC instances than the one checking DC ones. In average, each algorithm require an average execution time that can be 8 times greater than the average execution time required for checking positive instances having the same order.

Figure 8: Not Controllable Instances in group Size010



2.4 STNUBenchmark2020

This benchmark is composed of five sub-benchmarks (named as *Test 1* benchmarks in [10]). Each sub-benchmark is composed of two sets: one set of DC STNU instances and one set of non-DC ones.

Each sub-benchmark, characterized by a number $n \in \{500, 1000, 1500, 2000, 2500\}$ contains STNU instances (DC/NOT DC) generated randomly using it.univr.di.cstnu.util.STNURandomGenerator (see Section 1.2) with the following parameters

Number of nodes, n	$n \in \{500, 1000, 1500, 2000, 2500\}$
Number of lanes	5
Number of contingent links, k	$k = n/10$ (hence, $k = O(n)$)
Max absolute weight of ordinary edges	150
Max contingent range	$[0, 20]$
Probability of constraint among nodes in different lanes	0.40

For these parameter choices, each node (but the first and the last one of each lane) has two incoming edges and two outgoing edges in the same lane, as well as an average of 2.56 incident edges representing temporal-coordination constraints with nodes in other lanes. Activation time-points have no temporal constraint with nodes of other lanes because we preferred to derived them from the constraints incident to their contingent time-points. Temporal-coordination constraints are set in a way that avoids introducing negative circuits among a pair of nodes. Therefore, the number of edges is, on average, $3.28n - 1.28k - 10$; hence, $m = O(n)$. For each value of $n \in \{500, 1000, 1500, 2000, 2500\}$, the benchmark contains 200 DC networks

and 200 non-DC networks for a total of 2000 instances distributed in ten sub-benchmarks.

Moreover, in each sub-benchmark of DC instances there are 100 instances that are a copy of the first DC instances of the benchmark but where the number of contingent links is reduced to $\sqrt{n}$.

2.4.1 Experimental Evaluation

In CSTNU Tool library there are three different STNU DC checking algorithms:

1. **RUL20**: is the main algorithm presented in [10] as Algorithm 10.
2. **RUL$^-$**: is the algorithm presented in [11].
3. **Morris14**: is the algorithm presented in [12].

All such implementations are available as DC checking option in the class it.univr.di.cstnu.algorithms.STNU in CSTNU Tool library.

In the following we propose two diagrams that show the performances of the three algorithm using the STNUBenchmark2020 benchmark.

We used an Oracle JVM 8 having 8GB of heap memory on a Linux box with one Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz. The parameters for the Oracle Java Virtual Machine 1.8.0_144 were: `-Xmx8g`, and `-Xms8g`.

The execution times were collected by a Java program (`Checker`, proposed in our package) that allows one to determine the average execution time—and its standard deviation—of a DC checking algorithm applied to a set of instances.

Figure 9 and 10 display the average execution times of the three algorithms across all ten sub-benchmarks.

Figure 9: Benchmarks with DC Instances

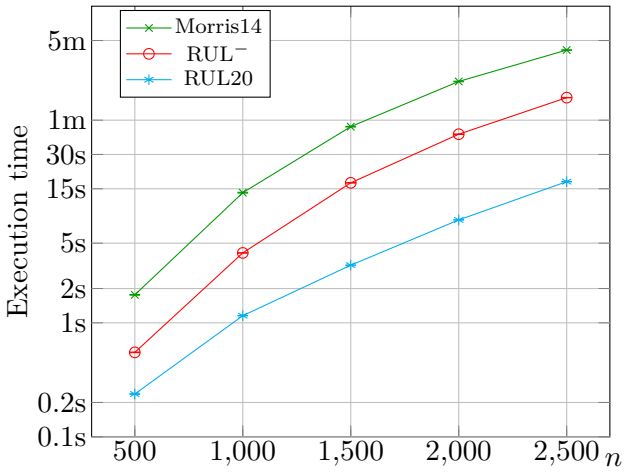
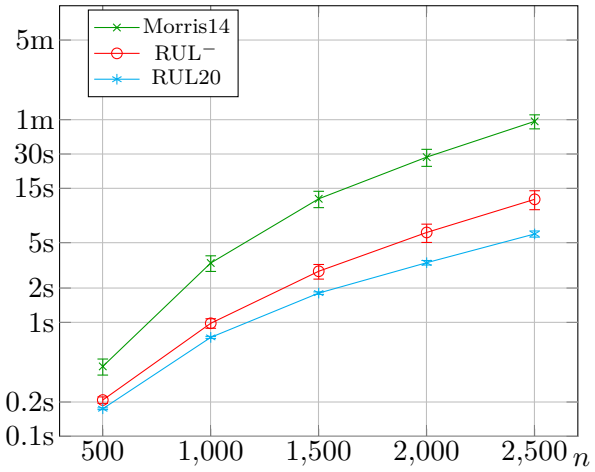


Figure 10: Benchmarks with non-DC Instances



Each plotted point represents average execution time over 200 instances

Each plotted point represents the average execution time for a given algorithm on the 200 instances of the given size, and the error bar for each point represents the 95% confidence interval. For example, over the 200 DC instances having $n = 2500$ time-points and $k = 250$ contingent links, the average execution time (in seconds) of the Morris14algorithm lies within the interval $[246.24, 248.56]$ with 95% confidence, while the average execution time of the RUL20algorithm lies within the interval $[17.26, 17.36]$ with 95% of confidence. These results demonstrate that the RUL20algorithm performs significantly better than the other two algorithms, especially over DC instances, but also over non-DC instances. For non-DC instances, the 95%-confidence intervals tend to be larger than those for the corresponding DC instances because for some non-DC instances the negative cycle can be detected immediately (e.g., by an initial run of Bellman-Ford or during the processing of the first contingent link or negative node), while others may require significant amounts of propagation.

One of our principal motivating hypotheses was that our new algorithm would be significantly faster than the RUL$^-$algorithm because it inserts significantly fewer new edges into the input STNU graph. In particular, whereas the RUL$^-$algorithm computes and inserts new edges arising from all three of the RUL$^-$rules, the RUL20algorithm only inserts edges arising from the length-preserving case of one rule.

2.5 CSTNPSUBenchmark2023

The structure of this benchmark is similar to the CSTNUBenchmark2020 one with two differences: the duration of each task is converted as guarded

adding two external bounds to the core. The external bounds enlarge the core of about 10% to each side. In particular, each contingent link (A, x, y, C) was replaced by the guarded link $(A, [[x', x][y, y']], C)$, where x' was set to $(1 - r)x$ while y' was set to $(1 + r)y$, where $r = .1$.

Since the scope of this benchmarks was to check the performance of the `PrototypalLink` algorithm, that is significant only for DC instances, the benchmarks contain only DC instances.

The following table summarizes the characteristics of each sub-benchmark.

Group	**Benchmark**	**instance indexes**	**#tasks**	**#XOR**	**#AND**	**CSTNPSU order**
Size010-3	B10-3-0	000-049	10	3	0	43
	B10-3-1	050-099	10	3	1	49
	B10-3-2	100-149	10	3	2	55
	B10-3-3	150-199	10	3	3	61
	B10-3-4	200-249	10	3	4	67

The total number of CSTNPSU instances is 250 DC, divided in 5 groups having 50 DC instances each.

2.5.1 Experimental Evaluation

This section presents an empirical evaluation of the performance of the FTNU DC-checking algorithm and of the `getPrototypalLink` procedure.

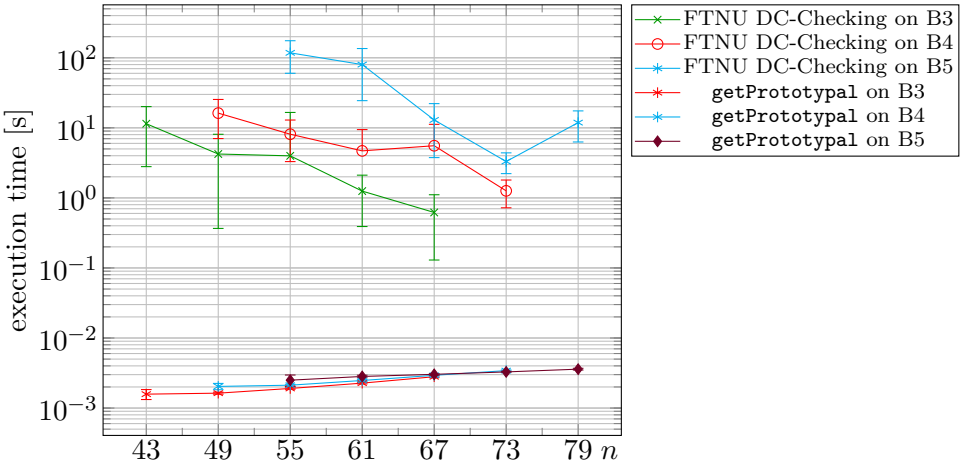
We recall that the `getPrototypalLink` procedure, given a DC instance as input, has to determine the completion of the instance and the path contingency span of each node to calculate the prototypal link.

The comparison of the two algorithm performances should give an idea of the computational cost for having a compact representation of a (sub)process versus the cost of determining its controllability only.

The tests were executed using a Java Virtual Machine 17 on an Apple PowerBook (M1 Pro processor) configured to use 8 GB memory as heap space.

Figure 11 displays the average execution times of the two algorithms over all five sub-benchmarks in B3, B4, and B5.

Figure 11: Benchmarks with DC Instances



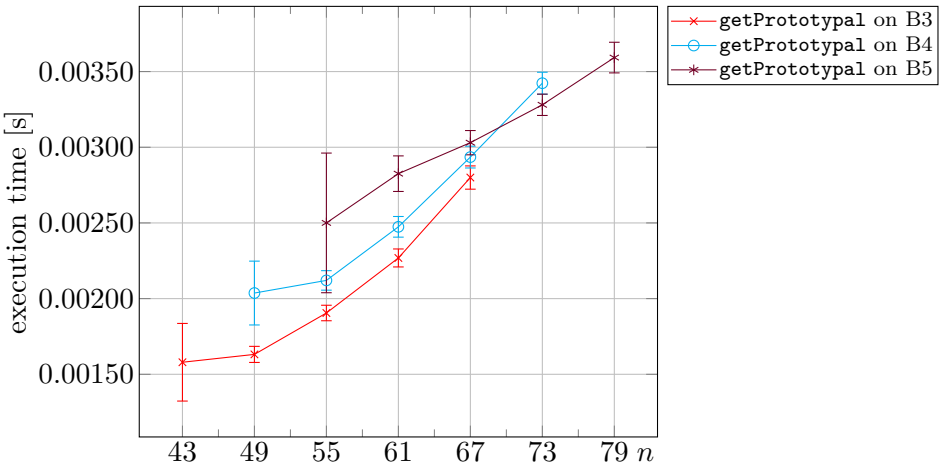
Each data point value is the sample average $\bar{X}_{50} = \frac{\sum_{i=1}^{50} X_i}{50}$ of average execution times X_i obtained considering the fifty instances of the relative sub benchmark. Indeed, each X_i is the average execution time obtained executing five times the algorithm on instance having index i in the considered sub benchmark. The error bar represents a 95% confidence interval for the average execution time of the algorithm on instances of the considered sub-benchmark.

As concerns the FTNU DC checking performance, from the data in Figure 11, it results that the performance is similar to the one obtained for the CSTNU DC checking algorithm in [13] although here the average times are one-order of magnitude smaller (thanks to the M1 processor). The more difficult instances are associated with workflows without parallel gateways (i.e., instances in the first sub-benchmark of each main benchmark) and the algorithm performs better as the number of AND gateways increases, but in B5. As stated in [14], such behavior is due to how the ATAPIS random generator works when the number of AND gateways is small (i.e., less than 5). Increasing the number of AND gateways (till 5), fewer XOR gateways are set in sequence and, therefore, there are fewer possible scenarios. In $B5_4$, where the number of AND gateways is 4, this pattern didn't occur. The sub-benchmark contains many instances with three-four observation timepoints over five in sequence, determining a greater number of possible scenarios and, hence, a greater execution time for the checking.

As concerns the `getPrototypalLink` procedure, its execution times are much lower than those of the DC checking algorithm (see Figure 11). Figure 12 shows the average execution time of `getPrototypalLink` of Figure 11 in linear y-scale. Once a network is checked DC, the completion phase updates the values of the original guarded and requirements links in the network while the building of the path contingency span graph creates and fills a vector of labeled distances from Z to each node. Such phases require visiting

each original edge of the network two times, each time considering all the labeled values associated with the edge. We verified that the average node degree is less than 5 in all benchmarks, hence the instances are sparse graphs. In these benchmarks, the quantity of labeled values present in each edge is not relevant as the number of edges/nodes in the determination of the computation time. Therefore, the `getPrototypalLink`-performance results to be quasi-linear with respect to the number of nodes.

Figure 12: `getPrototypalLink` Average Execution Time detail



References

[1] A. Lanz and M. Reichert, "Enabling time-aware process support with the atapis toolset," in *Proceedings of the BPM Demo Sessions 2014* (L. Limonad and B. Weber, eds.), vol. 1295 of *CEUR Workshop Proceedings*, pp. 41–45, 2014.

[2] C. Combi, M. Gambini, S. Migliorini, and R. Posenato, "Representing business processes through a temporal data-centric workflow modeling language: An application to the management of clinical pathways," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 44, pp. 1182–1203, Sept. 2014. doi:10.1109/TSMC.2014.2300055.

[3] Object Management Group (OMG), "Business process definition metamodel (bpdm), Beta 1." http://www.omg.org, 2007.

[4] L. Hunsberger, R. Posenato, and C. Combi, "A sound-and-complete propagation-based algorithm for checking the dynamic consistency of conditional simple temporal networks," in *22st International Symposium on Temporal Representation and Reasoning (TIME 2015)*, pp. 4–18,

IEEE, Sept. 2015. URL: http://dx.doi.org/10.1109/TIME.2015.26, doi:10.1109/TIME.2015.26.

[5] L. Hunsberger and R. Posenato, "Checking the Dynamic Consistency of Conditional Temporal Networks with Bounded Reaction Times," in *Proceedings of the 26th International Conference on Automated Planning and Scheduling, ICAPS 2016*, pp. 175–183, 2016. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13108.

[6] M. Cairo, L. Hunsberger, R. Posenato, and R. Rizzi, "A Streamlined Model of Conditional Simple Temporal Networks - Semantics and Equivalence Results," in *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, vol. 90 of *LIPIcs*, pp. 10:1–10:19, 2017. doi:10.4230/LIPIcs.TIME.2017.10.

[7] L. Hunsberger and R. Posenato, "Simpler and faster algorithm for checking the dynamic consistency of conditional simple temporal networks," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI-18*, pp. 1324–1330, International Joint Conferences on Artificial Intelligence Organization, July 2018. doi:10.24963/ijcai.2018/184.

[8] L. Hunsberger and R. Posenato, "Reducing epsilon-DC Checking for Conditional Simple Temporal Networks to DC Checking," in *25th International Symposium on Temporal Representation and Reasoning (TIME 2018)* (N. Alechina, K. Nørvåg, and W. Penczek, eds.), vol. 120 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 15:1–15:15, 2018. doi:10.4230/LIPIcs.TIME.2018.15.

[9] L. Hunsberger and R. Posenato, "Faster Dynamic-Consistency Checking for Conditional Simple Temporal Networks," in *30th Int. Conf. on Automated Planning and Scheduling, ICAPS 2020*, vol. 30, pp. 152–160, 2020. URL: https://www.aaai.org/ojs/index.php/ICAPS/article/view/6656.

[10] L. Hunsberger and R. Posenato, "A note on speeding up dc-checking for stnus," Tech. Rep. RR 109/2021, Department of Computer Science, University of Verona, July 2021. URL: https://iris.univr.it/handle/11562/1045707.

[11] M. Cairo, L. Hunsberger, and R. Rizzi, "Faster dynamic controllablity checking for simple temporal networks with uncertainty," in *25th Int. Symp. on Temporal Representation and Reasoning (TIME-2018)*, vol. 120, pp. 8:1–8:16, 2018. doi:10.4230/LIPIcs.TIME.2018.8.

[12] P. Morris, "Dynamic controllability and dispatchability relationships," in *Integration of AI and OR Techniques in Constraint Programming.*

CPAIOR 2014., vol. 8451 of *LNCS*, pp. 464–479, Springer, 2014. doi:10.1007/978-3-319-07046-9_33.

[13] L. Hunsberger and R. Posenato, "Sound-and-Complete Algorithms for Checking the Dynamic Controllability of Conditional Simple Temporal Networks with Uncertainty," in *25th Int. Symp. on Temporal Representation and Reasoning (TIME-2018)*, vol. 120, pp. 14:1–14:17, 2018. doi:10.4230/LIPIcs.TIME.2018.14.

[14] L. Hunsberger and R. Posenato, "Dynamic Controllability Checking for Conditional Simple Temporal Networks with Uncertainty: New Sound-and-Complete Algorithms based on Constraint Propagation," Tech. Rep. 105, Computer Science Department-University of Verona, Feb. 2018. URL: http://hdl.handle.net/11562/977720.