DECENT: A Benchmark for Decentralized Enforcement

Florian Gallay(✉) and Yliès Falcone

Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France
florian.gallay1@etu.univ-grenoble-alpes.fr

Abstract. DECENT is a benchmark for evaluating decentralized enforcement. It implements two enforcement algorithms that differ in their strategy for correcting the execution: the first one explores all alternatives to perform a globally optimal correction, while the second follows an incremental strategy based on locally optimal choices. Decent allows comparing these algorithms with a centralized enforcement algorithm in terms of computational metrics and metrics for decentralized monitoring such as the number and size of messages or the required computation on each component. Our experiments show that (i) the number of messages sent and the internal memory usage is much smaller with decentralized algorithms (ii) the locally optimal algorithm performs closely to the globally optimal one.

1 Introduction

Runtime enforcement consists in preventing the violation of a specification by using the so-called enforcers, which alter the execution whenever necessary. Conceptually the execution is typically abstracted as a trace, that is a sequence of system states. An enforcer takes as input such trace, modifies it *if needs be* to comply with the specification and then produces it as output. Existing enforcement frameworks are defined in the so-called centralized setting where there is a single enforcer acting on a global observation and control point.

As systems become increasingly decentralized (e.g. finance, vehicles, drone swarms), it is desirable to be able to ensure their critical properties while preserving their decentralization. In decentralized enforcement, a system consists of several components with one enforcer attached to each component. Since every enforcer can only observe what is happening locally, they need to communicate to gather information on the whole system and collaborate to modify the current execution.

In [13], we introduced two algorithms for the decentralized enforcement of properties specified using Linear-time Temporal Logic (LTL) [19]. Both of these algorithms are online algorithms that modify the current event if by appending this event to the trace output so far, this would violate the property. These algorithms differ by how they compute the corrected event. In the so-called *global* algorithm, all the possible alternatives for the emitted event are explored so that we are guaranteed to find the most *optimal* correction, whereas, in the so-called *local-incremental* algorithm, safe choices are made on each component in an incremental manner, without backtracking. However, these algorithms were not implemented nor evaluated.

T. Dang and V. Stolz (Eds.): RV 2022, LNCS 13498, pp. 293–303, 2022.
https://doi.org/10.1007/978-3-031-17196-3_18

This paper introduces DECENT, a simulation environment that implements the two decentralized enforcement algorithms to benchmark them against randomly generated formulas and patterns or using a specific formula. In DECENT, enforcement is performed offline on randomly generated traces as the goal is to get results on the performance of the algorithms independently of a system. In practice, however, our algorithms also work for online enforcement by reading the execution as it is produced by the system under scrutiny. The paper is structured as follows. Sect. 2 recalls the main principles of the approaches we presented previously. In Sect. 3, we overview the tool, and in Sect. 4 we discuss how we evaluated the algorithms, and we present the results. In Sect. 5, we discuss related work. Finally, we conclude in Sect. 6.

2 Principles of Decentralized Enforcement

We briefly overview the decentralized enforcement algorithms and refer to [13] for a formal definition. In the decentralized setting, multiple enforcers communicate to gather information. Whenever they observe an event, the enforcers use their internal memory to compute alternatives to the observed event in case of a violation. Common to the two enforcement algorithms are the following steps. The enforcers maintain a formula to enforce at any time, which is rewritten using *progression* [1] to separate the present from the future obligations. Then, in turn, the enforcers partially evaluate the formula using every possible assignment of their local atomic propositions while keeping track of the number of modifications compared to the original observed event. If one of the assignments leads to the partially evaluated formula being simplified to $\bot$, then the corresponding event is a violation and is removed from the memory of the enforcer.

The algorithms differ in their strategy to modify their local observation. In the global algorithm, once an enforcer is done evaluating the formula, it sends its memory to the next enforcer. Once the present obligations of the formula are entirely evaluated, the last enforcer applies a decision rule to pick which event to emit and communicates its decision to all the others. In the local-incremental algorithm, once an enforcer has evaluated the formula, it applies a local decision to send a single partial event (that may already be different from the observed event) to the next enforcer instead of the whole memory (which contains multiples events and the associated partially evaluated formulas). When the last enforcer is done, it applies the local decision rule to determine the event to emit. In this case, the last enforcer communicates the next formula to enforce as the other enforcers have already decided which event to emit locally.

We proved that the algorithms guarantee classical properties in enforcement: they are *sound*, meaning that the global output sequence of the enforcers does not violate the specification, and *transparent*, meaning that the global event is only modified if it does not comply with the specification. Additionally, the global algorithm is also *optimal* because the number of modifications to atomic propositions is minimal.

3 DECENT Overview

DECENT[1] implements the algorithms mentioned above using the functional programming language OCaml (in about 2200 LLOC). We reused some modules implemented

[1] https://gitlab.inria.fr/monitoring/decent.

in DECENTMON [2,3], mainly the implementation of LTL, events, traces and the associated generators. We implemented a few additional modules for the centralized and decentralized enforcement algorithms. The other functionalities of DECENTMON are left as is to allow monitoring or enforcement of formulas.

With DECENT, we can either parse specific formulas given in a file or randomly generate them. In both cases, formulas are enforced against a randomly generated trace using the decentralized algorithms as well as a "centralized" orchestration-based enforcement algorithm to compare them. When generating random formulas, it is possible to either specify a (maximum) size or to choose the specification patterns (defined in [6]) as templates to generate properties. Moreover, formula generation can be "biased" to place using more atomic propositions on a component. The underlying system is represented by an alphabet expressing how the atomic propositions are spread over the components. The alphabet is given as a command line argument, but it is also possible to use multiple different alphabets given in a file (to vary the distribution of the atomic propositions over the components or to add extra components/atomic propositions, for example). When generating traces (i.e. lists of events), each atomic proposition of the alphabet has a fixed probability of being included in each event (flip coin distribution), and it is possible to choose a different probability distribution such as Bernoulli, exponential or beta. Finally, DECENT offers two enforcement modes: *optimistic* or *pessimistic*. In the former, possible alternative local events are computed only when the observed event leads to a violation (i.e. after performing a round of verification first), while in the latter, local alternatives are always computed (i.e. at the same time as the verification). By default, the *pessimistic* mode is used.

4 Evaluating the Algorithms

We define the evaluation metrics in Sect. 4.1 and present the experiments to compare the algorithms in Sect. 4.2.

4.1 Metrics

The metrics we consider for this benchmark are mainly related to messages and the internal memory of the enforcers. We explain how measuring these metrics helps to understand the intricate behavior of enforcement algorithms.

Number of Modifications of the Events. As mentioned in Sect. 2, the local algorithm does not guarantee *optimality*. This metric allows us to observe how far is the local algorithm from optimality. It is worth noting that *optimality* is defined for each event individually when compared to the observed event, not for the whole trace. More specifically, it is possible (although quite rare from our experiments) for the total number of modifications in the enforced trace produced with the local algorithm to be lower than

it would have been using the global algorithm (on the same trace and formula). As the local algorithm relies on local information to decide the event to send to the next enforcer, it is impossible to guarantee that the two algorithms emit the same event. Consequently, the algorithms can end up with a different formula to enforce at some point (since the formula depends on the emitted event). Therefore, getting a stricter formula in the global version is possible in which the next observed event is a violation (forcing the enforcers to modify it). However, it is not in the local version. The number of modifications should, however, be identical between the global version and the centralized one because they both explore every alternative and should therefore be able to pick the same verdict every time. To see whether these differences are common, we also measured the number of differing events and the number of events with a different number of modifications between the enforced traces. Result tables show the average number of modifications in column **#mod**.

Number and Size of Messages. To measure the intensity of the communication required between monitors, we also measure the number (**#msg**) and the size of messages in the number of bytes used to encode them (**|msg|**). We only count the messages sent during the main evaluation process and not the ones sent after the final decision as their size depends on an implementation choice, that is, whether we send only the enforced event (which means the enforcers have to re-compute the next formula to enforce) or both the event and the next formula. In both cases, the last enforcer has to send a message to all the others, which causes the same number of extra messages as in the centralized version (the central enforcer also has to notify the local enforcers of the verdict). If we only send the event, the extra messages contain the same information as in the centralized version (so their size would be close if not identical). Otherwise, the final messages are larger in the decentralized version as they contain the next formula to enforce.

Size of the Temporal Correction Log. We also measure the size of the `tcl` (in bytes, given in **`|tcl|`**), i.e. the main internal structure used by the enforcers to compute the alternatives to compare the usage of the internal memory.

4.2 Experiments

For each experiment, we compare the performances of three algorithms (centralized “orchestration-based" [4], decentralized “global" and decentralized “local") using the same randomly generated traces and formulas (with some added constraints on the generation depending on the experiment). In all of them, we generate 1000 formulas and traces of size 100 (the size of a trace is the number of events composing it). We did not use larger traces because 1) it made the execution time of the experiments much longer while not changing much in the results but mainly, 2) it made the pathological cases much worse. We will talk about the latter in more detail with the first experiment. Also, we did not measure the execution time or the delays that would be caused by the

algorithms because the results would not be realistic as we do not take into account the cost of communication between the enforcers and because interacting with the system in a real application may have a significant impact on the overhead. We detail the two experiments in the following (an additional one can be found in Appendix A).

Varying Formula Size. Here, the generated formulas are of different sizes (from 1 to 6) to assess the scalability of our approach. As in DECENTMON, we use the maximum nesting of operator as a measure of size because it reflects well the difficulty to evaluate a formula. For example, $\mathbf{G}(a \vee b)$ has a size of 2 and $\mathbf{G}b \vee \mathbf{F}\neg a$ has a size of 3. Also, it is worth noting that we consider the size before we apply a simplification on the formula (so the simplified one could be smaller). We used the alphabet $\{a1, a2|b1, b2|c1, c2\}$, that is, there are 3 components in the system that can observe 2 atomic proposition each (for instance, the first component can observe $a1$ and $a2$).

Table 1. Results of the first experiment: varying formula size.

Algorithm	|φ|	#mod	#msg	|msg|	|tcl|	|φ|	#mod	#msg	|msg|	|tcl|
Cent.		0.434	3.	4.494	33.535		0.256	3.	4.496	284.037
Global	1	0.434	0.33	9.305	35.261	4	0.256	0.597	114.253	290.589
Local		0.429	0.258	4.087	27.845		0.252	0.448	40.996	151.169
Cent.		0.354	3.	4.496	57.655		0.242	3.	4.496	614.369
Global	2	0.354	0.403	23.296	60.801	5	0.242	0.609	196.24	630.496
Local		0.348	0.304	9.958	43.168		0.237	0.469	71.083	335.887
Cent.		0.313	3.	4.496	111.421		0.214	3.	4.493	2145.78
Global	3	0.313	0.463	48.572	117.324	6	0.214	0.777	386.229	2124.98
Local		0.311	0.357	20.256	76.424		0.211	0.618	108.125	635.299

Table 2. Differences between the enforced traces in the first experiment.

Variables	Average number of different events			Average number of events with a different #mod		
|φ|	Cent. $\leftrightarrow$ Glob.	Cent. $\leftrightarrow$ Loc.	Glob. $\leftrightarrow$ Loc.	Cent. $\leftrightarrow$ Glob.	Cent. $\leftrightarrow$ Loc.	Glob. $\leftrightarrow$ Loc.
1	0.	0.059	0.059	0.	0.	0.
2	0.	0.634	0.634	0.	0.	0.
3	0.	0.758	0.758	0.	0.026	0.026
4	0.	1.174	1.174	0.	0.057	0.057
5	0.	0.99	0.99	0.	0.022	0.022
6	0.	1.302	1.301	0.	0.075	0.075

Table 1 and 2 show the results of the first experiment. Here, the average number of modifications between the three version is almost identical. Although it is an average over the whole trace, it at least means that, in practice, the local algorithm is very close to the other versions (even though it does not guarantee *optimality*). It is worth noting that #mod sometimes has a lower value for the local algorithm. This could be caused either by what we mentioned in the previous section or because, in some cases, we can reach a state where the next formula to enforce is $\top$ (at which point we stop the enforcement). As we cannot guarantee that the local algorithm outputs the same trace as the others, it may miss this state and enforce a few more events before reaching it. Because of this, as we do the average of #mod on the whole trace, if the trace given by the local version is larger, it skews the average by a little bit. Aside from this, the cost in terms of local memory usage (|tcl|) is greatly reduced by the local algorithm (upwards of almost 3 times smaller with formulas of size 6) and the number of messages (#msg) is much smaller with the decentralized algorithms. There are fewer messages because the enforced formulas do not necessarily contain every atomic proposition of the alphabet, and therefore, some enforcers may not need to do any work. However, the size of messages (|msg|) is much larger than in the centralized algorithm because they contain more information (alternatives and the associated partially evaluated formulas vs only the local observations in the centralized version). It is worth noting that messages with the local version are much smaller than with the global one (about 2 to 4 times depending on the size of the formula). Moreover, the traces produced by the centralized and the global algorithm are identical. The traces given by the local version are almost identical, with about one event different from the other two. This supports our claim that the local version is very close to being *optimal* despite not guaranteeing it.

An issue of this experiment is that some formulas are particularly bad for these methods because the rewriting to separate their present and future obligations cause an explosion of their size. As we generate some random formulas, we may get one of these pathological cases for only one size and not the others, which would majorly impact the results for this size. Typically, pathological cases are formulas with many Until (**U**) and Globally (**G**) operators interleaved so they are quite uncommon for smaller formulas and we have not seen any using specification patterns (i.e. the next experiment). As we mentioned earlier, these formulas are one of the reasons why we chose to use traces of this size (with larger traces, the execution would take way too much time to complete as the formula size tends to increase after each event in these cases).

Using Realistic Specifications. In this experiment with realistic specifications, we used formulas generated with the LTL specification patterns [6] (we omitted universality response chain because of size constraint, the complete tables are available on the tool repository). Here, we also compare the two enforcement modes *optimistic* and *pessimistic* (on the same formulas/traces by setting a seed). We used the same alphabet as in the previous experiment (Table 4).

Results of the second experiment: using specification patterns

Table 3. Pessimistic algorithm

Algorithm	Pattern	#mod	#msg	|msg|	|tcl|
Cent.	abs	0.37	3.	4.502	180.414
Global		0.37	0.629	79.477	181.993
Local		0.372	0.472	39.586	116.784
Cent.	exist	0.162	3.	4.503	201.799
Global		0.162	0.684	77.222	199.828
Local		0.162	0.463	40.489	119.418
Cent.	bexist	0.242	3.	4.502	743.064
Global		0.242	0.586	389.644	809.429
Local		0.242	0.392	211.063	732.576
Cent.	prec	0.149	3.	4.502	347.696
Global		0.149	1.087	136.227	350.039
Local		0.14	0.63	55.954	164.459
Cent.	resp	0.039	3.	4.501	637.256
Global		0.039	1.084	252.875	618.055
Local		0.039	0.729	100.551	337.767
Cent.	pchain	0.191	3.	4.503	640.231
Global		0.191	1.037	218.258	634.57
Local		0.19	0.656	90.923	334.564
Cent.	cchain	0.193	3.	4.504	1441.1
Global		0.193	1.255	491.208	1362.17
Local		0.204	0.955	163.378	623.357

Table 4. Optimistic algorithm

Algorithm	Pattern	#mod	#msg	|msg|	|tcl|
Cent.	abs	0.37	3.	4.502	158.365
Global		0.37	0.586	51.782	165.268
Local		0.372	0.583	43.809	117.334
Cent.	exist	0.162	3.	4.503	188.028
Global		0.162	0.532	49.569	198.448
Local		0.162	0.529	45.389	152.229
Cent.	bexist	0.242	3.	4.502	243.993
Global		0.242	0.418	281.677	307.038
Local		0.242	0.417	272.871	293.366
Cent.	prec	0.149	3.	4.502	433.583
Global		0.149	0.74	72.624	443.767
Local		0.14	0.706	65.051	198.991
Cent.	resp	0.039	3.	4.501	945.261
Global		0.039	0.759	129.307	1026.3
Local		0.039	0.753	121.769	630.653
Cent.	pchain	0.191	3.	4.503	709.02
Global		0.191	0.777	122.98	722.777
Local		0.19	0.758	108.346	392.53
Cent.	cchain	0.193	3.	4.504	1706.91
Global		0.193	1.114	244.29	1691.65
Local		0.204	1.115	197.499	755.796

Table 5. Differences between the enforced traces in the second experiment.

Variables	**Average number of differing events**			**Average number of event with a different #mod**		
Pattern	**Cent. ↔ Global**	**Cent. ↔ Local**	**Global ↔ Local**	**Cent. ↔ Global**	**Cent. ↔ Local**	**Global ↔ Local**
abs	0.	1.805	1.805	0.	0.005	0.005
exist	0.	0.276	0.276	0.	0.	0.
bexist	0.	0.009	0.009	0.	0.001	0.001
prec	0.	0.946	0.946	0.	0.005	0.005
resp	0.	0.257	0.257	0.	0.001	0.001
pchain	0.	0.655	0.655	0.	0.01	0.01
cchain	0.	3.974	3.974	0.	0.274	0.274

Tables 3 to 5 give the results of this experiment. There is still an improvement in memory usage (`|tcl|`) using the local algorithm. However, it is not as large as in the previous experiment (about two times smaller at most instead of three previously, and the same observation applies to the message sizes). A notable difference with the *pessimistic* mode is that, for some patterns, there seems to be a large difference in the number of messages between both decentralized algorithms: for instance, with the *precedence* pattern, there were almost twice as many messages sent in the global version on

average. Using the *optimistic* mode, the memory usage is improved for some patterns (e.g. *bounded existence*) but it is also worse for others (e.g. *response*). It seems that this mode is better for certain types of formulas although it is hard to tell exactly because our metrics are not well suited to compare both. For example, with $|\texttt{tcl}|$, we only include in the average the results of runs where enforcement was required. Otherwise, it would skew the result and make it seem like *optimistic* is significantly better, which is untrue as the (enforcement) cost is identical to the other mode when enforcement is required (and null when it is not). However, this metric largely depends on the enforced formula and has consequently a large variance over the different runs. Therefore, if the runs that required enforcement are the ones where $|\texttt{tcl}|$ was large, then the average will be worse (or at best similar) than what we get with the *pessimistic* mode, even though we might have gained a lot overall in computation time by not doing useless enforcement steps. The main drawback of this mode is that the overhead is larger when enforcement is required because of the initial verification phase. Table 5 shows that, for some patterns, the local algorithm produces traces that have more differences compared to the other two versions than with random formulas (mainly constrained chain where we measured 4 different on average). We have not included a table showing the difference between the enforced traces with the *optimistic* mode because the values are identical (i.e. the exact same traces are produced).

5 Related Work

There are many approaches to tackling the problem of *decentralized monitoring* (see [12] for an overview). The closest approaches to ours are the ones using formula rewriting on LTL [2,3,22]. Other methods use different formalisms to express the specifications like finite-state automata [9] or Stream Runtime Verification (SRVs) [5] or have different assumptions on the system like in monitoring decentralized specifications [7], that is having an independent specification for each component instead of one for the whole system. The aforementioned approaches have been implemented in various tools such as DECENTMON [2,3,9], using Maude [22], dLola [5] or THEMIS [7,8]. All these approaches only perform verification: they report violations or satisfaction of a property and do not consider enforcement.

On the topic of *runtime enforcement*, most of the work has been done for centralized systems with varying assumptions about the underlying system: specification expressed with discrete-time formalisms [10,11], timed properties [17] or with uncontrollable events in the system [16,20]. We note that fewer methods have been actually implemented for centralized enforcement, see TiPEX [18] or GREP [21] for instance. Although there are a few approaches [14,15] considering decentralized enforcement in decentralized systems, these are tailored to specific systems and our work is the first generic approach able to enforce any property expressed in linear-temporal logic.

6 Conclusions

DECENT allows the evaluation of two decentralized enforcement algorithms introduced in [13]. Our experiments demonstrate that, although the messages are much larger with

the decentralized algorithms, we need less of them. The internal memory usage is significantly reduced with the local-incremental algorithm. Also, even though the latter algorithm does not guarantee the *optimality*, it is very close to it in practice.

We plan to implement our algorithms into a larger tool (THEMIS, for instance) which would allow us to use these algorithms in real systems. We also plan to improve the algorithms as they suffer from a few limitations, mainly the dependence on a powerful simplification function (to not miss any verdict) and the exponential blowup of the formulas in some rare cases. As both of these limitations come from LTL and rewriting, we plan to use different specifications formalism such as finite-state automata or even a more expressive one like timed properties or streams. Finally, we also plan to study in greater detail the *optimistic* and *pessimistic* modes to try and find settings in which one method is better than the other using more suited metrics.

A Third experiment

We observe how the performance evolves when we add more components to the system and/or add atomic propositions to the alphabet. Here, the generated formulas are of fixed size (6, chosen arbitrarily). We study six different systems where components can observe either 1, 2 or 3 atomic proposition(s) each and with 3 or 5 components. The number of observable atomic propositions is indicated by $|\Sigma_i|$ and the number of components by $|\Sigma|$.

Table 7 and 6 show the results of this experiment. The difference in size of the messages between both decentralized versions seems to get larger when the system gets larger (in particular when adding more atomic propositions to each component). This makes sense as the global version explores every possible alternative and a larger system has more of them. The local version is not affected as much because a local decision is applied before an enforcer communicates with the next one to limit the growth of the internal memory (we can also observe this when looking at |`tcl`|) which,

Table 6. Differences between the enforced traces in the third experiment

Algorithm	$\lvert\Sigma_i\rvert$	$\lvert\Sigma\rvert$	**#mod**	**#msg**	**|msg|**	**|`tcl`|**	$\lvert\Sigma_i\rvert$	$\lvert\Sigma\rvert$	**#mod**	**#msg**	**|msg|**	**|`tcl`|**
Cent.			0.2	3.	3.5	608.58			0.183	5.	3.5	2441.84
Global	1	3	0.2	0.771	236.249	636.719	1	5	0.183	1.093	537.952	2503.89
Local			0.196	0.609	108.197	401.75			0.177	0.838	168.346	739.348
Cent.			0.202	3.	4.501	8765.14			0.221	5.	4.502	2941.78
Global	2	3	0.202	0.759	1250.89	8967.43	2	5	0.221	1.098	534.934	2952.26
Local			0.198	0.58	322.203	2703.01			0.219	0.839	115.972	580.382
Cent.			0.205	3.	5.754	3707.75			0.177	5.	5.757	11485.
Global	3	3	0.205	0.76	614.68	3705.48	3	5	0.177	1.091	1133.39	11466.
Local			0.203	0.609	124.239	859.064			0.172	0.837	236.039	1396.81

Table 7. Results of the third experiment: varying system size

Variables		Average number of different event			Average number of event with a different #mod		
$\lvert\Sigma_i\rvert$	$\lvert\Sigma\rvert$	Cent. ↔ Global	Cent. ↔ Local	Global ↔ Local	Cent. ↔ Global	Cent. ↔ Local	Global ↔ Local
1	3	0.	1.264	1.264	0.	0.015	0.015
1	5	0.	1.147	1.147	0.	0.05	0.05
2	3	0.	1.16	1.16	0.	0.059	0.059
2	5	0.	1.459	1.459	0.	0.065	0.065
3	3	0.	0.951	0.951	0.	0.029	0.029
3	5	0.	1.377	1.377	0.	0.084	0.084

in turn, limits the growth of the messages as they contain the internal memory. Adding components to the system seems to increase the average memory usage in the global and centralized versions which makes sense as there are more alternatives to consider in this case as well. Table 6 suggests that the system size does not have a major impact on the differences between the traces produced by the algorithms.

References

1. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Ann. Math. Artif. Intell. **22**(1–2), 5–27 (1998)
2. Bauer, A., Falcone, Y.: Decentralized LTL Monitoring. Technical report, March 2012. 31 pages
3. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. In 14th International Conference on Runtime Verification, Toronto, Canada, September 2014
4. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods Syst. Des. **49**(1), 109–158 (2016)
5. Danielsson, L.M., Sánchez, C.: Decentralized stream runtime verification. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 185–201. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_11
6. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002), pp. 411–420 (1999)
7. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Bultan, T., Sen, K. (eds.) Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, 10–14 July, 2017, pp. 125–135. ACM (2017)
8. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: ISSTA 2017. Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 125–135. Santa Barbara, United States (2017)
9. Falcone, Y., Cornebize, T., Fernandez, J.-C.: Efficient and generalized decentralized monitoring of regular languages. In: Ábrahám, E., Palamidessi, C. (eds.) FORTE 2014. LNCS, vol. 8461, pp. 66–83. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43613-4_5
10. Falcone, Y., Fernandez, J.-C., Mounier, L.: What can you verify and enforce at runtime? Int. J. Softw. Tools Technol. Transf. **14**(3), 349–382 (2012)

11. Falcone, Y., Mounier, L., Fernandez, J.-C., Richier, J.-L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods Syst. Des. **38**(3), 223–262 (2011)
12. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 176–210. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
13. Gallay, F., Falcone, Y.: Decentralized LTL enforcement. In: GandALF 2021–12th International Symposium on Games. Automata, Logics, and Formal Verification, pp. 1–18. Padua, France (2021)
14. Hallé, S., Khoury, R., Betti, Q., El-Hokayem, A., Falcone, Y.: Decentralized enforcement of document lifecycle constraints. Inf. Syst. **74**(Part), 117–135 (2018)
15. Hu, C., Dong, W., Yang, Y., Shi, H., Deng, F.: Decentralized runtime enforcement for robotic swarms. Front. Inf. Technol. Electron. Eng. **21**(11), 1591–1606 (2020). https://doi.org/10.1631/FITEE.2000203
16. Khoury, R., Hallé, S.: Runtime enforcement with partial control. In: Garcia-Alfaro, J., Kranakis, E., Bonfante, G. (eds.) FPS 2015. LNCS, vol. 9482, pp. 102–116. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30303-1_7
17. Pinisetty, S.: Runtime enforcement of timed properties revisited. Formal Methods Syst. Des. **45**(3), 381–422 (2014). https://doi.org/10.1007/s10703-014-0215-y
18. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena Timo, O.: Runtime enforcement of timed properties revisited. Formal Methods Syst. Des. **45**(3), 381–422 (2014). https://doi.org/10.1007/s10703-014-0215-y
19. Pnueli, A.: The temporal logic of programs. In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pp. 46–57. IEEE Computer Society, 1977
20. Renard, M., Falcone, Y., Rollet, A., Jéron, T., Marchand, H.: Optimal enforcement of (timed) properties with uncontrollable events. Math. Struct. Comput. Sci. **29**(1), 169–214 (2019)
21. Renard, M., Rollet, A., Falcone, Y.: GREP: games for the runtime enforcement of properties. In: Yevtushenko, N., Cavalli, A.R., Yenigün, H. (eds.) ICTSS 2017. LNCS, vol. 10533, pp. 259–275. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67549-7_16
22. Rosu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. **12**(2), 151–197 (2005)