GREP: Games for the Runtime Enforcement of Properties

Matthieu Renard, Antoine Rollet, Yliès Falcone

GREP: Games for the Runtime Enforcement of Properties

Matthieu Renard[1], Antoine Rollet[1], and Yliès Falcone[2]

[1] LaBRI, Bordeaux INP, University of Bordeaux, Bordeaux, France
first.last@labri.fr
[2] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr

Abstract. We present GREP, a tool for the runtime enforcement of (timed) properties. GREP takes an execution sequence as input (*stdin*), and modifies it (*stdout*) as necessary to enforce the desired property, when possible. GREP can enforce any regular timed property described by a deterministic and complete Timed Automaton. The main novelties of GREP are twofold: it uses game theory to improve the synthesis of enforcement mechanisms, and it accounts for *uncontrollable* events, i.e. events that cannot be controlled by the enforcement mechanisms and thus have to be released immediately. We present an overview of GREP and validate its usability with a performance evaluation.

1 Runtime Enforcement of Timed Properties with Uncontrollable Events

Runtime Verification (RV, [12,10,5]), also referred to as passive testing ([8,2]), consists in checking if the execution of a running system satisfies some given specification. Unlike static verification, RV studies a real execution of a system, possibly after deployment. This paper deals with *runtime enforcement* (RE, [19,13,11,6]), an extension of runtime verification where executions are corrected when they violate the desired property (see [9] for an overview). An enforcement mechanism (EM) modifies the execution of a running system: it takes an execution as input and outputs a possibly-different execution. One of the advantages of RE is that the whole specification of the system under scrutiny is not necessary to generate an EM, only a property that should be satisfied by its output is needed. The general scheme is given in Fig. 1. We distinguish two categories of actions: *controllable* actions which can be modified by an enforcement mechanism, and *uncontrollable* actions which can only be observed by the enforcement

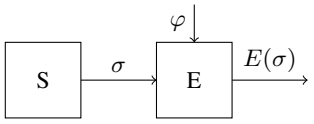


Fig. 1: Schematic description of an enforcement mechanism E, modifying the execution σ of the system S to $E(\sigma)$, so that it satisfies the property φ.

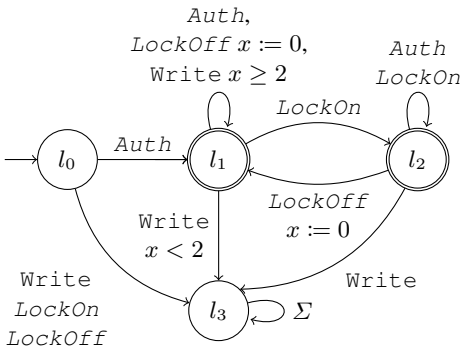


Fig. 2: Property φ_{t} modelling writes on a shared storage device ([17])

mechanism. Enforcement mechanisms should be *sound*, *compliant* and *optimal*, meaning that the output should satisfy the specification when possible, the output should be as close to the input as possible, and the output should be maximal, respectively. In [17,16], we introduce runtime enforcement for timed properties with uncontrollable events, and we propose a game approach for generating the EM in [18]. A comparison of this approach with related work may also be found in [16]. GREP implements the game approach of [18] extended to timed properties.

1.1 Timed Properties and Automata

In this paper, properties are modelled with regular timed properties described by Timed Automata (TA) [4]. Consider the following property on a simple shared storage device: after Authentication, a user can write a value only if the storage has been unlocked for at least 2 time units. (Un)locking the device is decided by another party, meaning that it is not controllable by the user. This property is formalised by the TA φ_{t} given in Fig. 2, with the alphabet of uncontrollable actions $\{\mathtt{LockOn}, \mathtt{LockOff}, \mathtt{Auth}\}$. In φ_{t}, the set of locations is $L = \{l_0, l_1, l_2, l_3\}$; the initial location is l_0; the set $X = \{x\}$ made of a single clock x is used to model time; the alphabet of all actions is $\Sigma = \{\mathtt{Auth}, \mathtt{LockOn}, \mathtt{LockOff}, \mathtt{Write}\}$; the set of accepting locations is $G = \{l_1, l_2\}$; and the set of transitions Δ contains, for example, $(l_1, x \geq 2, \mathtt{Write}, \emptyset, l_1)$. A transition is composed of an initial location, a guard, an action, a set of clocks to reset, and a target location. For instance, the transition $(l_1, x \geq 2, \mathtt{Write}, \emptyset, l_1)$ means that location l_1 is reached from location l_1 if the $\mathtt{Write}$ action occurs when the clock x is greater than or equal to 2, with no clock to reset; and $(l_2, \top, \mathtt{LockOff}, \{x\}, l_1)$, means that l_1 is reached from l_2 when the $\mathtt{LockOff}$ action occurs, resetting clock x while taking the transition.

1.2 Description of the approach

The strategy of GREP is based on the enforcement approach proposed in [16,18]. As shown in Fig. 1, an EM may be seen as a function from timed words to timed words,

with the ability to delay some controllable actions using a memory, but with no possibility to act on uncontrollable actions. This mechanism should ensure the correctness of the output sequence (*soundness*), that the order of controllable actions is preserved and that uncontrollable actions are immediately released (*compliance*), and that the output is maximal (*optimality*). All these definitions, as well as the proofs of correctness and details of the EM generation, at two levels of abstraction (functional and operational), are provided in [16]. Notice that unlike the approach in [15], there could be some situations where the property may not be enforceable, since uncontrollable actions cannot be retained.

Let us provide the intuition of the approach on an example. Consider the required property φ_t given in Fig. 2, and the input sequence $(1, \texttt{Auth})\,(1, \texttt{LockOn})\,(2, \texttt{Write})$ $(1, \texttt{LockOff})$ $(1, \texttt{LockOn})$ $(1, \texttt{Write})$ $(1, \texttt{LockOff})$. Table 1 gives the evolution of the system at different time instants, providing the output of the EM at a given date, its complete expected output if no other event is received (the output at an infinite date), and the state of the memory (the stored controllable actions) at an infinite date. At $t = 1$ and $t = 2$ respectively, `Auth` and `LockOn` actions are received. Since they are uncontrollable actions, they are released immediately. At $t = 4$, action `Write` is received. Since it is controllable and in order to avoid to reach a bad state, it is stored in the memory. At $t = 5$, the uncontrollable action `LockOff` is received and immediately released. Now it is possible to emit the stored `Write` action, but only after 2 time units. For this reason, $(2, \texttt{Write})$ is added at the end of σ_s meaning that `Write` should be emitted in 2 time units, and it is removed from σ_c. At $t = 6$, another uncontrollable `LockOn` action is received and immediately released. At this step, it is not possible to emit `Write` anymore, then it is removed from the end of σ_s and put back at the beginning of σ_c. At $t = 7$, another controllable `Write` action is received and stored (added at the end of σ_c). At $t = 8$, the last uncontrollable `LockOff` action is received, allowing to emit the two `Write` actions after 2 time units. Thus, they are placed at the end of σ_s and removed from σ_c. Since no other action is received afterwards, the two `Write` actions are released at $t = 10$.

1.3 Games

In order to improve the computation time of the EM at runtime, it is possible to precompute the behaviour of the EM ahead of the execution and storing it. For this purpose, we use game theory. GREP builds a two-player game graph representing the possible actions of the EM and the system under scrutiny (that acts as the environment). Each vertex of the graph belongs to one of these two players, and each edge represents a possible action of the player that owns the source vertex. GREP then solves a Büchi game by computing a set of nodes of the graph from which there exists a winning strategy. More details may be found in [18]. Using this approach allows us to avoid the exploration of the whole execution tree at runtime.

More precisely, GREP proceeds in three steps: first, it computes a symbolic graph, which is a finite abstraction of the (infinite) semantics of the TA; then it computes the game graph and the winning strategy for the EM; finally the EM follows the strategy to enforce the property.

Table 1: Table showing the evolution of the enforcement mechanism with input $(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(2, \texttt{Write})$ $(1, \texttt{LockOff})$ $(1, \texttt{LockOn})$ $(1, \texttt{Write})$ $(1, \texttt{LockOff})$ over time.

t	Output	Complete Expected Output	Buffer
1	$(1, \texttt{Auth})$	$(1, \texttt{Auth})$	ϵ
2	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$	ϵ
4	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$	$\texttt{Write}$
5	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(2, \texttt{Write})$	ϵ
6	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(1, \texttt{LockOn})$	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(1, \texttt{LockOn})$	$\texttt{Write}$
7	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(1, \texttt{LockOn})$	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(1, \texttt{LockOn})$	$\texttt{Write}$ $\texttt{Write}$
8	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(1, \texttt{LockOn})$ $(2, \texttt{LockOff})$	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(1, \texttt{LockOn})$ $(2, \texttt{LockOff})$ $(2, \texttt{Write})$ $(0, \texttt{Write})$	ϵ
10	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(1, \texttt{LockOn})$ $(2, \texttt{LockOff})$ $(2, \texttt{Write})$ $(0, \texttt{Write})$	$(1, \texttt{Auth})$ $(1, \texttt{LockOn})$ $(3, \texttt{LockOff})$ $(1, \texttt{LockOn})$ $(2, \texttt{LockOff})$ $(2, \texttt{Write})$ $(0, \texttt{Write})$	ϵ

The symbolic graph abstracting the semantics of the TA is similar to the usual zone graph used to compute reachability on TAs (see for example [7]), except that the successor relation is more constraining. In the usual zone graph, it is only required that a state in a vertex (vertices can be seen as sets of states of the semantics of the TA that share the same location) can reach a state in the successor vertex, whereas for our purpose it is required that all the states in a vertex can reach a state in the successor vertex. This holds for delay transitions (transitions representing elapse of time) and action transitions (representing a transition in the TA) which are built in the same way. We also require that each vertex has at most only one time successor (i.e. a node reached by letting time elapse, as opposed to nodes reached by outputting events (action successors)). An algorithm to compute such a graph is given in [3], for example.

The game graph is then built upon this symbolic abstraction graph. Each vertex of the symbolic abstraction graph is duplicated, one of the resulting vertices belongs to player 0 (the EM), and the other to player 1 (the environment). Each of these vertices is then associated with words taken from a finite, prefix-closed set of controllable words, each vertex being duplicated again for every word from this set. This set of controllable words is computed to ensure that the strategy is always the best one (the one outputting the maximal word) using only these vertices. For a formal description of this set in case of untimed properties, see [18]. The edges of the game graph represent the actions of the player: an edge leaving a node belonging to player 0 represents either the emission of the first event of the associated word (which represents the stored controllable actions), leading to a vertex belonging to player 0 again, since an EM can output multiple events at the same time, or to the same node which belongs to player 1, meaning that the EM decides not to emit for the moment and lets the environment play. An edge leaving a vertex belonging to player 1 leads to a vertex belonging to player 0, and represents

either the reception of an uncontrollable event (changing the symbolic state to an action successor), the reception of a controllable event (changing the associated word), or the elapse of time (changing the symbolic state to the time successor). Nodes that are their own time successor (i.e. that contain all the valuations reached by letting time elapse) have their corresponding edge replaced by an edge leading to the same vertex which belongs to player 0. This corresponds to receiving no more event, thus allowing us to consider finite inputs, but with infinite plays over the graph (since there will be a loop between a node belonging to player 0 where the EM will decide not to emit anything, and the environment not receiving any event and being stable by elapse of time). A formal definition of the game graph for untimed properties can be found in [18] too. GREP builds a game graph which is similar, except that some nodes have time successors.

After having constructed the game graph, GREP computes the winning set of nodes of the Büchi game with Büchi nodes (the nodes to be always reachable) defined as all the vertices of the game graph whose symbolic location of a vertex is an accepting location of the TA for player 0.

Then, GREP can follow a real execution on the game graph, by watching the node that has been reached so far by its output, and the nodes that can be reached by emitting stored controllable actions (i.e. following the corresponding edges in the game graph). Whenever a winning node is reached by player 0, the strategy is to emit as many events as possible, remaining in a winning node all the time. Since the winning nodes are winning a Büchi game, it is always possible for player 0 to stay in a winning node whenever one is reached. Whenever a winning node is reached, the output of the EM is then guaranteed to satisfy the property.

An example of game graph associated with property φ_{t} given in Fig. 2 is provided in Fig. 3. In this graph, nodes are labelled with a tuple (l, z, w, p), where l is the location of the TA associated with the node, z is the zone (set of clock constraints), w is the word of controllable actions as described previously, and p is the player: 0 for the enforcement mechanism, and 1 for the environment. The square node is the initial node, the blue (rectangular) ones are the winning nodes, and the nodes that are double-circled are the nodes whose locations are accepting. Note that in our example, all the accepting nodes (double-circled) are winning (blue, rectangular). The edges have different colours and heads, depending on their role: green edges (filled triangular heads) correspond to the enforcer emitting its first stored controllable action, blue ones (empty triangular heads) to the enforcer not emitting (thus letting the environment play), red (filled diamond heads) and orange (empty diamond heads) edges correspond to the reception of an uncontrollable event and a controllable one, respectively, and purple ("vee" heads) edges represent the elapse of time (they lead to the time successor).

2 General Description of GREP

GREP is a tool of about 6,000 lines of code[3] developed using the C language, available at `https://github.com/matthieurenard/GREP`. GREP is essentially composed of 2 modules (cf Fig. 4): Symbolic Computing Module (SCM) and Enforcement

[3] calculated with cloc (`https://github.com/AlDanial/cloc`)

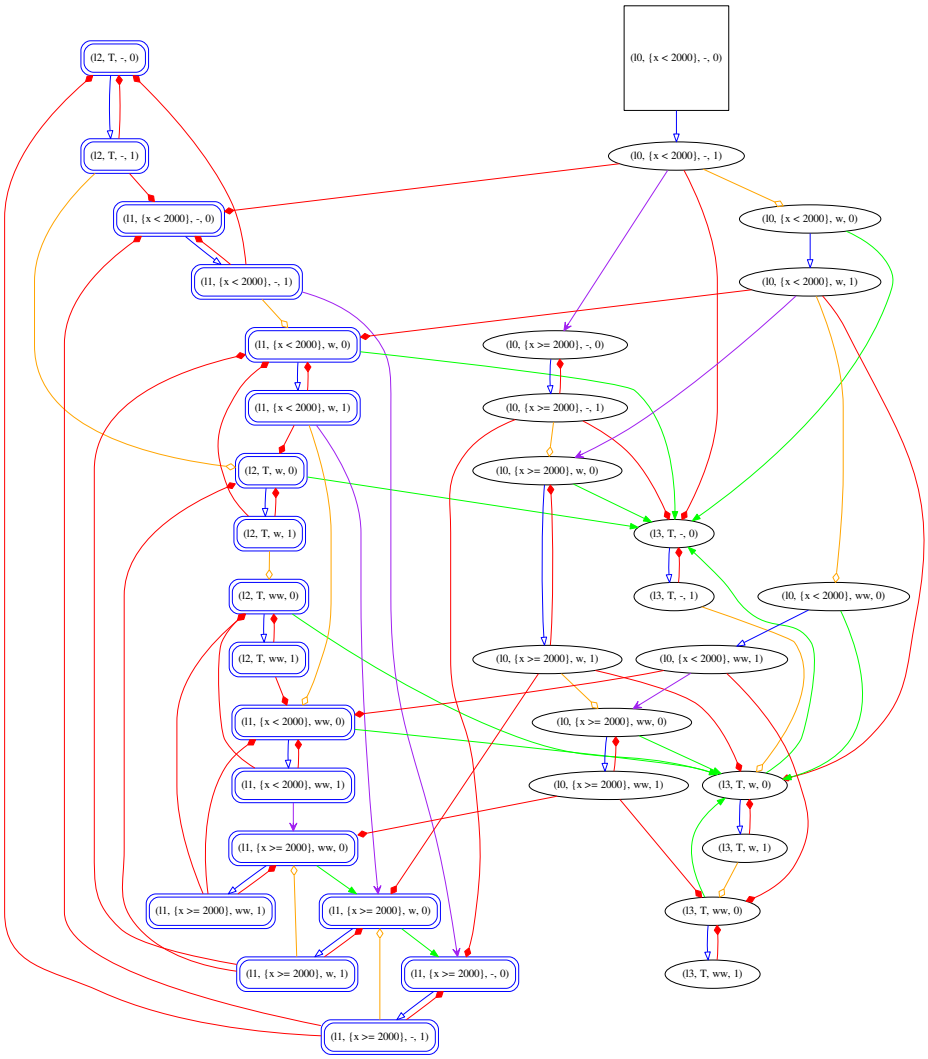


Fig. 3: Game graph associated with property φ_{t}

Monitor Module (EMM). It loads a TA file describing the desired property, and reads the inputs directly from *stdin*. The output of the EM is sent to *stdout*. This approach allows one to use GREP with off-the-shelf applications.

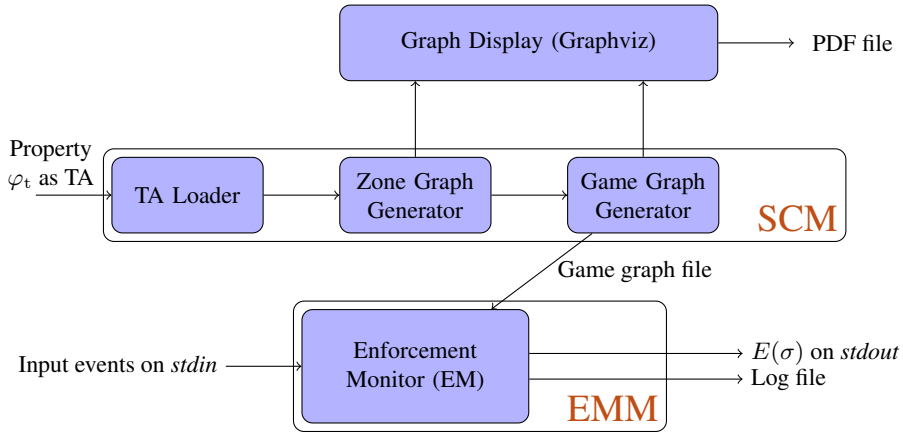


Fig. 4: General architecture of GREP

2.1 Symbolic Computing Module (SCM)

The Symbolic Computing Module is composed of three main components: a TA loader, the zone graph generator, and the game graph generator.

TA Loader The TA loader is the component that parses a file containing the description of a timed automaton and loads it into a C structure. The file of the automaton is a textual description following a grammar designed for this purpose. The automaton must be also deterministic and complete (see [4]). If the automaton is not deterministic, the behaviour is undefined. Once the timed automaton is loaded, a symbolic graph is computed by the Zone Graph Generator to abstract its infinite semantics into a finite graph.

Zone Graph Generator From the timed automaton, a symbolic graph is constructed using zones. Compared to a classical zone graph (used to compute reachability), this symbolic graph satisfies additional constraints. (A zone graph usually requires that, between a node and its successor, there exists a state of the semantics of the timed automaton in the first node that leads to a state in its successor). In this symbolic graph, all states in a node must lead to the successor. An algorithm to compute this symbolic graph satisfying these constraints is given in [3]. This algorithm has been implemented to compute the symbolic graph in this module. In GREP, zones are represented by Difference Bound Matrices (DBMs), using the UPPAAL DBM library (udbm, [1]), and its C API. The algorithm requires some functionality that is not provided by this C API

(some of them exist in some higher-level wrappers), such as complementing zones into a list of zones. This functionality was added to our own wrapper of udbm. No other third-party library was needed to compute the symbolic graph. This symbolic graph is used to build the final game graph, that will be used by the enforcement monitor.

Game Graph Generator Using the symbolic graph, the Game Graph Generator builds a graph over which to play a Büchi game whose strategy is the one to be followed by the enforcement monitor. The graph is constructed as described in [18], adding some edges to represent the elapse of time (that changes zones). Once the graph is constructed, the Büchi game is solved for player 0 (the enforcement monitor), with the set of Büchi nodes being the set of nodes whose location is accepting. The winning nodes are the nodes from which the enforcement monitor ensures that its output will satisfy the property. Following a path of winning nodes in the graph gives a strategy to follow such that the final output satisfies the property. This is how the EM uses the graph to actually enforce the property.

2.2 Enforcement Monitor Module (EMM)

The EMM module uses SCM to compute the output for a given input. It has five main public functions: $\mathrm{init}(G)$, $\mathrm{getStrat}()$, $\mathrm{delay}(t)$, $\mathrm{eventRcvd}(e)$, and $\mathrm{emit}()$. Function $\mathrm{init}(G)$ initialises the EMM following the strategy from graph G. Function $\mathrm{getStrat}()$ gives the strategy to follow, i.e. whether the first action of the buffer should be output or not. Since time is abstracted by the zone graph for the SCM, the SCM needs to be notified that some time has passed, which is done by the use of function $\mathrm{delay}(t)$, where t is the number of time units that have elapsed since the last call to delay, or the creation of the enforcement mechanism for the first call. Time units only need to be consistent with the ones used in the property. Function $\mathrm{eventRcvd}(e)$ is used to inform the EMM that an event e has been read from the input. In this case, the EMM acts differently depending on the controllability of the event. Function $\mathrm{emit}()$ is used to output the first action of the buffer (uncontrollable events are output by function $\mathrm{eventRcvd}()$, as required by compliance).

Note that these functions allow to use the EMM in both online (real-time) and offline (with a trace as input) settings. All these functions, except function $\mathrm{getStrat}()$, return the number of time units required to reach the time successor of the current node (∞ if there is no time successor). It is the number of time units given to function $\mathrm{delay}()$ if no event is received before and the strategy is not to emit.

Thus, the general algorithm to use the EMM in the offline setting is given in Algorithm 1. Basically, the EMM follows a path in the game graph. Thus, it considers the current node as the node reached by its output, and explores the strategy tree from this node. The EMM also stores the controllable actions that have not been output yet, and uses them to compute the possible output. Since the output should be the longest possible, with minimal possible delays, computing the strategy requires to explore the tree of all possible strategies. This is done by exploring the game graph, simulating the emission of the controllable actions of the buffer at all possible time instants. In each node belonging to player 0, if the successor by emitting (green, empty triangular head

```
input : The game graph 𝒢, the input sequence of events, through function read()
output: The output of the enforcer mechanism, through function emit()
1  init(G);
2  del ← ∞;
3  while The input sequence has not been read entirely do
4  |  (δ, a) ← read();
5  |  while del ≤ δ do
6  |  |  δ ← δ − del;
7  |  |  del ← delay(del);
8  |  |  while getStrat() = EMIT do
9  |  |  |  emit();
10 |  |  end
11 |  end
12 |  delay(δ);
13 |  del ← eventRcvd(a);
14 end
15 while del < ∞ or getStrat() = EMIT do
16 |  while getStrat() = EMIT do
17 |  |  emit();
18 |  end
19 |  if del < ∞ then
20 |  |  del ← delay(del);
21 |  end
22 end
```

Algorithm 1: Main algorithm to enforce a property in offline mode

arrow in the game graph) is winning, then it is explored, and if the time successor is also winning, it is explored as well, since waiting before emitting could allow the EMM to output more events. Each node is then associated with a score, corresponding to the number of actions that have been emitted to reach the node. Then, the EMM stores the node that has the biggest score, and the strategy to follow to reach it. If two nodes have the same score, then the lowest common ancestor is computed, and the one node that can be reached by emitting from this ancestor (the other node can be reached from this ancestor by waiting) is kept as the node to reach (this corresponds to computing the lexicographical order). This process is repeated for each node with the same score, with the previous stored node, such that in the end the stored node is the minimal node (for the lexicographic order) of all the nodes with the highest score.

Note that computing an output such that all actions are emitted whenever it is possible to emit them does not require to explore the strategy tree. Depending on the property, the two outputs could be the same (i.e. if the property is such that letting time elapse never enables a transition that eventually allows the EMM to output more events), thus the EMM can work faster by using an optimisation that does not compute any tree, but outputs actions whenever possible (i.e. when the successor node by emitting is winning) if it is specified to do so.

2.3 User Interface

GREP is shipped with two executables: one to use the enforcement mechanism in offline mode, and the other in the online mode. Both of them take their input on the standard input. In the offline mode, the input is composed of events in the form (t, a), where t is a date and a is an action, controllable or uncontrollable. In the online mode, only the action is given, the date is computed from the real time through a call to `gettimeofday()`. Note that these executables may build only on UNIX-like systems because of some system calls such as `gettimeofday()` and `clock_gettime()`. Excepting this, the tool is not system-dependent. The output (events with their dates) is printed on the standard output. Several options may be used:

- One of the two options `-a <automatonFile>` or `-g <graphFile>` must be passed to specify the property. The file `<automatonFile>` should be in the same format as the file shown in appendix A. The file `<graphFile>` should be a file saved by this executable (see option `-s`), loading this kind of file should be faster than loading an automaton file since it contains the graph, which does not need to be computed again.
- `-s <graphFile>` saves the game graph in `<graphFile>`, to be loaded in another execution (see option `-g`).
- `-z <zoneGraphFile>` draws the zone graph using graphviz and store it (as PDF) in `<zoneGraphFile>`.
- `-d <gameGraphFile>` draws the game graph using graphviz and store it (as PDF) in `<gameGraphFile>`.
- `-t <timeFile>` logs times between the reception of two events in the file `<timeFile>`. This option is used to benchmark the program.
- `-l <logFile>` prints all the logs in `<logFile>`.
- `-f (fast)` use the optimised version, where actions are output whenever they can be instead of outputting the longest word possible with minimal dates.

If options `-s`, `-z`, `-d`, or `-t` are not given, then the corresponding action will just not happen. For example, without `-z`, the zone graph will not be saved. If none of the options among `-a` and `-g` is given, the program will print an error and abort. If both are given, then the automaton file is used. If option `-l` is not given, then the standard error is used as log file, which is not recommended (we recommend always using the option `-l`). If the option `-f` is not given, then the enforcement mechanism will output as many events as possible, with the lowest possible dates; enabling the option will make it output actions if it is possible to output them (i.e. if the node of the game graph reached by outputting is winning). Using option `-f` is usually faster, but the outputs might differ depending on the property. Example usage:

```
game_enf_offline -a phiext.tmtn -l log \
    -d gameGraph.pdf < input
```

will enforce the property described in the file `phiext.tmtn`, logging in the file `log`, reading its events from the file `input`. It will also draw the game graph in the file `gameGraph.pdf`.

The enforcement mechanism logs the mode in which it runs (default or fast) at the beginning, and when it stops, it logs the input, its output, the controllable actions that

have not been output (what remains in its buffer), and a verdict that is WIN if its output satisfies the property, or LOSS otherwise (some properties might not be enforced as explained in [17]).

```
Enforcer initialized in default mode.
Shutting down the enforcer...
Summary of the execution:
Input: (0, Write) (1, Auth) (2, Write) (3, LockOn)
       (4, Write) (5, LockOff) (6, LockOn) (7, LockOff)
Output: (1, Auth) (2, Write) (2, Write) (3, LockOn)
       (5, LockOff) (6, LockOn) (7, LockOff) (9, Write)
Remaining events in the buffer:
VERDICT: WIN
Enforcer shutdown.
```

Listing 1.1: Example log file produced by GREP

For example, considering that `phiext.tmtn` is the file given in appendix A, the previous command with the input file containing the sequence:
$(0, \texttt{Write})$ $(1, \textit{Auth})$ $(2, \texttt{Write})$ $(3, \textit{LockOn})$ $(4, \texttt{Write})$ $(5, \textit{LockOff})$ $(6, \textit{LockOn})$ $(7, \textit{LockOff})$, produces the output:
$(1, \textit{Auth})$ $(2, \texttt{Write})$ $(2, \texttt{Write})$ $(3, \textit{LockOn})$ $(5, \textit{LockOff})$ $(6, \textit{LockOn})$ $(7, \textit{LockOff})$ $(9, \texttt{Write})$. The produced log file is given in Listing 1.1.

3 Performance Evaluation

The performance of GREP has been evaluated on three properties that come along with TiPEX, the tool to which we compare. TiPEX (see [14]) is, to our knowledge, the only other tool that acts as an enforcement mechanism for timed regular properties. These properties are described in Fig. 5. The safety property states that there should always be 5 time units between two r actions. The co-safety property states that the first r action should be followed by a g action, with a delay of at least 6 time units. The response property states that every grant (g) action should be followed by a release (r) action within 15 to 20 time units, without any grant action occurring between them.

For each of these properties, GREP has been run 100 times on every input among 100 inputs of 1000 events randomly generated. The time between the reception of two events has been saved for all of these executions. The same times have been computed for TiPEX[4], reducing the number of inputs and iterations to have the benchmarks run in a reasonable amount of time. Figures 6 and 7 give a graphical visualisation of the performances of GREP and TiPEX.

Figures 6 and 7 are obtained as follows: each input is iterated several times (100 for GREP, less for TiPEX), and the computation times of the tool between the reads of two consecutive events of the input are stored. Then, the median time is computed for each of these times between all the iterations. We then plot the logarithm (in base 10) of these times against the reads of the events. We use a logarithmic scale in nanoseconds because

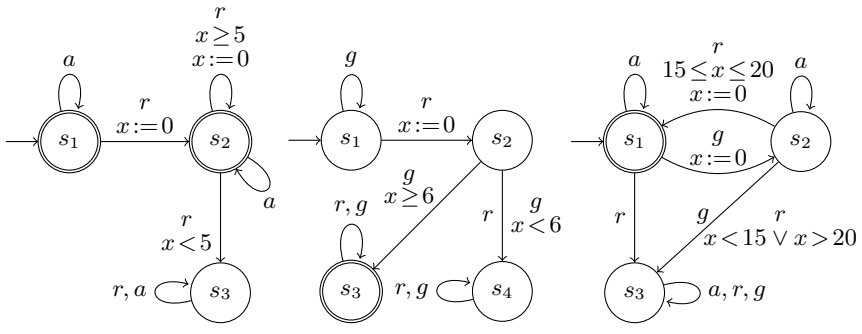
[4] We patched TiPEX to retrieve the times as we do in our tool, only modifying it to get times properly, and did not change the behaviour inside the part that is being measured.

many values are low, and they would be merged in a line when using a linear scale. The results for GREP with option `-f` are given only for the safety property because they are similar to the results without the option for the two other properties. We can see that GREP is faster than TiPEX by several orders of magnitude. GREP outputs many events in less than $10\,\mu s$ (4 on the graphs), whereas TiPEX takes at least 1 ms (6 on the graph) to output them. For the safety property, we can see that for some inputs, GREP takes an increasing amount of time to compute the strategy. This is due to the exploration of the strategy tree, which grows with the number of stored controllable actions. Using the optimised setting (`-f`) allows GREP to compute its output faster, as shown in Fig. 6b. The last vertical line has also many high values, because it represents the time to emit all the remaining actions after the last event from the input was read. For the co-safety and response properties, GREP is less variable than for the safety property, mainly because its strategy is simpler: it consists in either emitting everything in the co-safety (once state s_3 is reached) or emitting nothing for the response property, if the first stored controllable is an r while in state s_1. TiPEX, on the other hand, takes a linearly-increasing amount of time to emit some events.

4 Discussion and Concluding Remarks

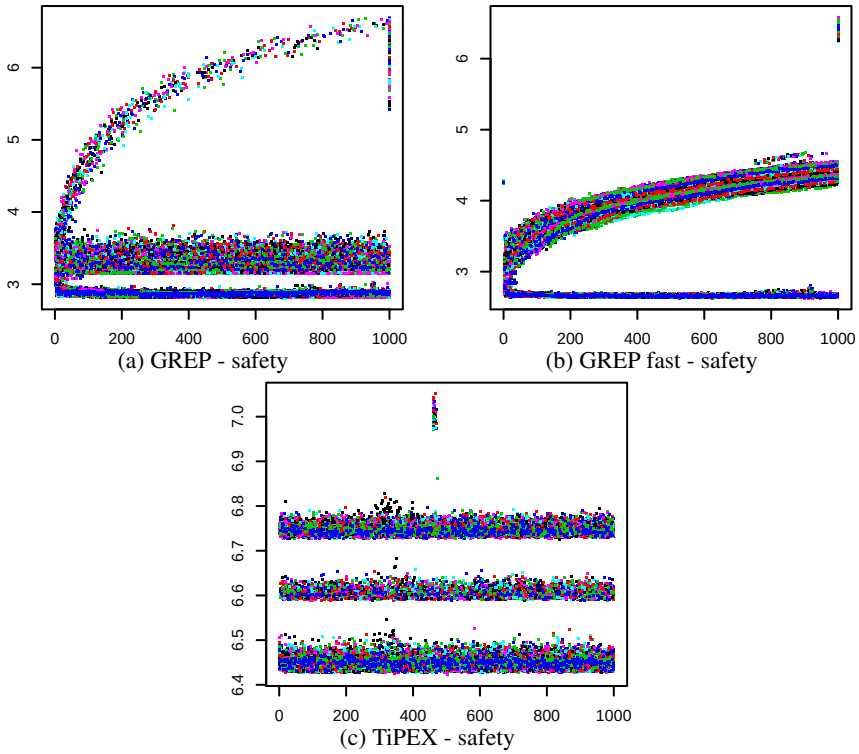
As shown in Section 3, GREP provides better computing times than TiPEX. There are several factors that can explain this. The implementation language is one of these factors: GREP is implemented in C, which produces assembly code that is fast to execute, whereas TiPEX is implemented in Python, which is a higher-level language that can introduce some overhead in the execution time. Another factor is the use of games to enforce the properties, that allows the EMM to compute the output faster. Indeed, the game graph allows us to know if a node is winning in very little time, where the same computation not using graph needs to consider all the reachable states with the buffer.

Note that our tool was designed primarily to handle uncontrollable events. The properties used in our evaluation/comparison do not feature uncontrollable events because TiPEX only supports controllable events. To our knowledge, there is no other tool that



(a) A safety property (b) A co-safety property (c) A response property

Fig. 5: Properties used to benchmark GREP



(a) GREP - safety (b) GREP fast - safety

(c) TiPEX - safety

Fig. 6: Comparison of timings of GREP and TiPEX on the safety property. "GREP fast" means that option `-f` is used. The x axis corresponds to the events of the input (from 1 to 1000), and the y axis corresponds to the logarithm of the timings (in nanoseconds) between the reads of the events.

acts as an enforcement mechanism for timed properties with uncontrollable events. We initially used games to precompute (with the game graph) the behaviour of the enforcement mechanism upon the reception of uncontrollable events, but it has also improved the computation time of the output even without any uncontrollable event.

Depending on the property, there could be an exponential blow-up in the number of nodes that have to be visited. The `-f` option allows the enforcement mechanism to use a strategy (where each event is output as soon as possible) that prevents this blow-up. To further improve the performance of GREP and avoid the blow-up when using the strategy outputting the longest words, we aim at i) computing a better set of controllable words to be associated with nodes, and having a better representation of controllable words. A (more theoretical) possible extension to this work is to determine under which conditions the two strategies (using the optimised version or not) are equivalent.

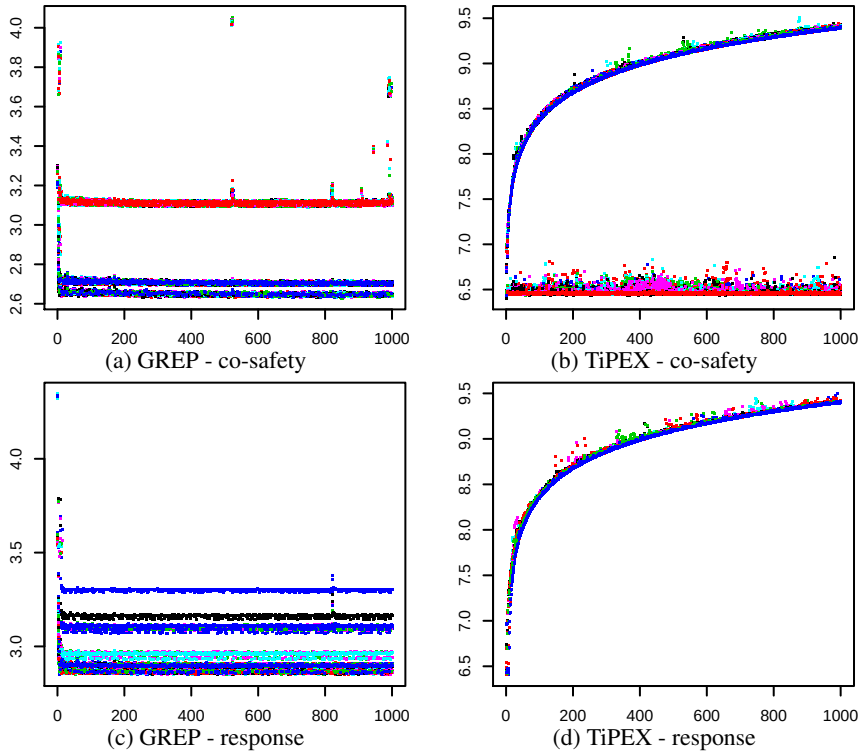


Fig. 7: Timings of GREP and TiPEX on the response and co-safety properties. The x axis corresponds to the events of the input (from 1 to 1000), and the y axis corresponds to the logarithm of the timings (in nanoseconds) between the reads of the events.

References

1. Uppaal DBM Library. http://people.cs.aau.dk/~adavid/UDBM/, accessed: 2017-04-27
2. Alcalde, B., Cavalli, A., Chen, D., Khuu, D., Lee, D.: Network protocol system passive testing for fault management: A backward checking approach. In: International Conference on Formal Techniques for Networked and Distributed Systems. pp. 150–166. Springer (2004)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Dill, D., Wong-Toi, H.: Minimization of timed transition systems. In: Cleaveland, W. (ed.) CONCUR '92: Third International Conference on Concurrency Theory Stony Brook, NY, USA, August 24–27, 1992 Proceedings, pp. 340–354. Springer Berlin Heidelberg, Berlin, Heidelberg (1992), http://dx.doi.org/10.1007/BFb0084802
4. Alur, R., Dill, D.: The theory of timed automata. In: de Bakker, J., Huizing, C., de Roever, W., Rozenberg, G. (eds.) Real-Time: Theory in Practice, Lecture Notes in Computer Science, vol. 600, pp. 45–73. Springer Berlin Heidelberg (1992)
5. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools,

and final results of crv 2014. International Journal on Software Tools for Technology Transfer pp. 1–40 (2017)
6. Basin, D., Jugé, V., Klaedtke, F., Zălinescu, E.: Enforceable security policies revisited. ACM Trans. Inf. Syst. Secur. 16(1), 3:1–3:26 (Jun 2013), http://doi.acm.org/10.1145/2487222.2487225
7. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets, pp. 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
8. Cavalli, A., Gervy, C., Prokopenko, S.: New approaches for passive testing using an extended finite state machine specification. Information and Software Technology 45(12), 837–852 (2003)
9. Falcone, Y.: You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 89–105. Springer (2010)
10. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems, pp. 141–175 (2013)
11. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design 38(3), 223–262 (2011)
12. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
13. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12(3), 19:1–19:41 (Jan 2009)
14. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: TiPEX: A Tool Chain for Timed Property Enforcement During eXecution. In: Bartocci, E., Majumdar, R. (eds.) RV'2015, 6th International Conference on Runtime Verification. Lecture Notes in Computer Science, vol. 9333, p. 12. Springer, Vienne, Austria (Sep 2015)
15. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.L.: Runtime enforcement of timed properties. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7687, pp. 229–244. Springer (2012)
16. Renard, M., Falcone, Y., Rollet, A., Jéron, T., Marchand, H.: Optimal enforcement of (timed) properties with uncontrollable events. Mathematical Structures in Computer Science p. 1–46 (2017)
17. Renard, M., Falcone, Y., Rollet, A., Pinisetty, S., Jéron, T., Marchand, H.: Enforcement of (timed) properties with uncontrollable events. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015. Lecture Notes in Computer Science, vol. 9399, pp. 542–560. Springer International Publishing (2015)
18. Renard, M., Rollet, A., Falcone, Y.: Runtime enforcement using Büchi games. In: Proceedings of Model Checking Software - 24th International Symposium, SPIN 2017, Co-located with ISSTA 2017, Santa Barbara, USA. pp. 70–79. ACM Press (July 2017)
19. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (Feb 2000)

A Automaton File

The automaton file describing φ_t (see Fig. 2) follows:

```
automaton {
    cont {Write}
    uncont {Auth, LockOn, LockOff}
    nodes {
        l0[initial];
        l1[accepting];
        l2[accepting];
        l3;
    }
    clocks {x}
    edges {
        l0 ->{Auth}{}{} l1;
        l0 ->{Write}{}{} l3;
        l0 ->{LockOn}{}{} l3;
        l0 ->{LockOff}{}{} l3;
        l1 ->{LockOn}{}{} l2;
        l1 ->{Write}{}{x >= 2} l1;
        l1 ->{LockOff}{x}{} l1;
        l1 ->{Auth}{}{} l1;
        l1 ->{Write}{}{x < 2} l3;
        l2 ->{Auth}{}{} l2;
        l2 ->{LockOn}{}{} l2;
        l2 ->{LockOff}{x}{} l1;
        l2 ->{Write}{}{} l3;
        l3 ->{Write}{}{} l3;
        l3 ->{Auth}{}{} l3;
        l3 ->{LockOn}{}{} l3;
        l3 ->{LockOff}{}{} l3;
    }
}
```