Model-Driven Trace Diagnostics for Pattern-based Temporal Specifications

Wei Dou
University of Luxembourg
Luxembourg
dou@svv.lu

Domenico Bianculli
University of Luxembourg
Luxembourg
domenico.bianculli@uni.lu

Lionel Briand
University of Luxembourg
Luxembourg
lionel.briand@uni.lu

ABSTRACT

Offline trace checking tools check whether a specification holds on a log of events recorded at run time; they yield a verification verdict (typically a boolean value) when the checking process ends. When the verdict is false, a software engineer needs to diagnose the property violations found in the trace in order to understand their cause and, if needed, decide for corrective actions to be performed on the system. However, a boolean verdict may not be informative enough to perform trace diagnostics, since it does not provide any useful information about the cause of the violation and because a property can be violated for multiple reasons.

The goal of this paper is to provide a practical and scalable solution to solve the trace diagnostics problem, in the settings of model-driven trace checking of temporal properties expressed in *TemPsy*, a pattern-based specification language. The main contributions of the paper are: a model-driven approach for trace diagnostics of pattern-based temporal properties expressed in *TemPsy*, which relies on the evaluation of OCL queries on an instance of a trace meta-model; the implementation of this trace diagnostics procedure in the TemPsy-Report tool; the evaluation of the scalability of TemPsy-Report, when used for the diagnostics of violations of real properties derived from a case study of our industrial partner. The results show that TemPsy-Report is able to collect diagnostic information from large traces (with one million events) in less than ten seconds; TemPsy-Report scales linearly with respect to the length of the trace and keeps approximately constant performance as the number of violations increases.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

OCL, temporal constraints, trace diagnostics, offline trace checking, run-time verification, specification patterns

ACM Reference Format:
Wei Dou, Domenico Bianculli, and Lionel Briand. 2018. Model-Driven Trace Diagnostics for, Pattern-based Temporal Specifications. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239396

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239396

1 INTRODUCTION

Run-time verification (RV) [31] is a verification technique used for checking the correctness of *an execution of a system* with respect to a specification. The checking procedure (represented by a *monitor* program) can be performed during the actual system execution (a settings called *online monitoring*) or by analyzing a log of recorded events produced by the system (a settings called *offline monitoring* or *offline trace checking*).

Among the many approaches for run-time verification [4], our previous work on model-driven trace checking [17]—developed as part of a research project in collaboration with our public-service partner CTIE (Centre des technologies de l'information de l'Etat, the Luxembourg national center for information technology)—was focused on model-driven run-time verification of business processes [16]. Our approach checks properties expressed in *TemPsy* (Temporal Properties made easy) [11, 17], a pattern-based domain-specific language for the specification of temporal requirements. The approach, implemented in the TemPsy-Check tool [18], relies on an optimized mapping of temporal requirements written in *TemPsy* into Object Constraint Language (OCL) constraints on a meta-model of execution traces. More specifically, it reduces the problem of checking a temporal property over an execution trace to the evaluation of an OCL constraint (derived from the property to check and semantically equivalent to it) on an instance of the trace meta-model.

Run-time verification tools (more specifically, monitors) yield a *verification verdict* (for short, verdict) after checking a property over an execution trace: the verdict is a truth value from some truth domain [31]. In the case of offline trace checking, typically the verdict is boolean (true/false). When the verdict is false, a software engineer needs to diagnose the property violations found in the trace in order to understand their cause and, if needed, decide for corrective actions to be performed on the system. However, *a boolean verdict may not be informative enough to perform trace diagnostics*. For example, let us consider the following temporal property, informally stated in English: "it is always the case that if event *A* occurs then it should stimulate, within 5 time units, the sequential occurrence of events *B* followed, within 2 time units, by *C*"; it corresponds to a time-constrained response chain pattern [2]. This property can be violated for various reasons, such as:

- there is at least an occurrence of *A* not followed by the sequence of events *B*-*C*;
- there is at least an occurrence of *A* that is followed by the

sequence of events *B*-*C* but *B* occurs after more than 5 time units since the last occurrence of *A*;
- there is at least an occurrence of *A* that is followed by the sequence of events *B*-*C* but *C* occurs after more than 2 time units since the occurrence of *B*.

Just yielding a boolean verdict for reporting a violation in the example above would be inadequate, since a boolean value in this case does not provide any useful information about the cause of the violation.

Despite their intrinsic limitations, boolean verdicts are used by state-of-the art tools [5, 7, 10, 33, 37] for offline trace checking, including our own TemPsy-Check. Some tools also pinpoint the last log entry (i.e., the last event) read before discovering the violation; however, the usefulness of this information is limited, because the last read event might not necessarily be the event responsible for the violation or, as shown above, because a requirement could be violated in different ways.

The goal of this paper is to provide a practical and scalable solution *to solve the trace diagnostics problem*, in the settings of model-driven trace checking of pattern-based temporal properties. Similarly to our previous work on model-driven trace checking [17], such a solution is constrained by the requirements determined by the type of context in which this work is set: R1) to be viable in the long term, any procedure shall rely on standard MDE (model-driven engineering) technology—in our context tools implementing OMG specifications; R2) any procedure shall be scalable and enable diagnostics of violations in large traces within practical time limits, such that the violations in a trace with millions of events could be processed within seconds.

Requirement R1 comes from our industrial partner, which uses a software development methodology that requires all solutions to adhere to OMG specifications. Although originating from the necessities of our partner, we assume that this requirement can be generalized to other contexts in which MDE is a mainstream practice within the software development process. Requirement R2 originates from the need to process arbitrarily sized logs within practical time limits, to make the use of trace diagnostics an enabler for performing timely corrective actions.

The trace diagnostics technique proposed in this paper leverages the pattern-based, flat (i.e., without operator nesting) structure of temporal properties expressed in *TemPsy* for the precise characterization of the violations that can occur with each type of property. Based on this characterization, the key idea is to retrieve, from a trace that violates a property, the diagnostic information that describes the violations admitted by the specific type of property. The technique follows a model-driven approach, based on the trace meta-model defined in our previous work [17]: it uses OCL *queries* defined on the trace model to analyze violations and collect diagnostic information. The queries are supported by a set of auxiliary OCL functions and optimized based on the structure of the targeted *TemPsy* property for achieving better performance.

We have implemented the approach proposed in this paper in a publicly available tool [12, 13]. Furthermore, we have developed an interactive visualization tool that provides a graphical visualization of the diagnostic information collected by TemPsy-Report, to support a better understanding of violations.

We evaluated the scalability of our TemPsy-Report tool by studying how the execution time varies depending on the trace length, the number of violations to diagnose, and the violation and property type. We used a benchmark of *TemPsy* properties based on real requirements extracted from our case study, on traces with length ranging from 100K to 1M. The results show that TemPsy-Report is able to collect diagnostic information from large traces (with one million events) in less than ten seconds; it scales linearly with respect to the length of the trace and keeps approximately constant performance as the number of violations increases.

To summarize, the main contributions of this paper are: i) a model-driven approach for trace diagnostics of pattern-based temporal properties expressed in *TemPsy*, which relies on the evaluation of OCL queries on an instance of a trace meta-model; ii) a publicly available tool TemPsy-Report, implementing the trace diagnostics procedure; iii) an evaluation of the scalability of TemPsy-Report, when applied to the diagnostics of violations of real properties derived from a case study of a complex information system developed and used by the Luxembourg government.

The rest of the paper is structured as follows. Section 2 provides some background information about the *TemPsy* language. Section 3 presents a classification of violations for the patterns supported by *TemPsy*. Section 4 illustrates TemPsy-Report, our model-driven procedure for trace diagnostics. Section 5 describes tool support. Section 6 reports on the evaluation of the scalability of the TemPsy-Report tool. Section 7 discusses related work. Section 8 concludes the paper and gives directions for future work.

2 BACKGROUND: THE *TEMPSY* LANGUAGE

TemPsy [11, 17] is a pattern-based, domain-specific language for the specification of temporal properties. It has been developed based on the analysis of the requirements of various applications implementing business process models in the context of eGovernment systems [16]. The analysis established that all the requirements of the case study could be expressed as temporal properties by using the property specification patterns proposed by Dwyer et al. [19], with some additional expressions. The resulting language supports the eight *patterns* ("absence", "universality", "existence", "bounded existence", "precedence", "response", "precedence chain", "response chain") and the five *scopes* ("globally", "before", "after", "between-and", "after-until") defined in [19]; patterns represent high-level abstractions of formal specifications while scopes indicate the portions of a system execution in which a certain pattern should hold. The new extensions introduced by *TemPsy* are: (1) the possibility, in the definition of a scope boundary, to refer to a specific occurrence of an event; (2) the possibility to indicate a time distance with respect to a scope boundary; (3) support for expressing time distance between occurrences in the *precedence* and *response* patterns (hereafter collectively called *order* patterns) as well as their chain versions; (4) additional variants for the bounded existence and absence patterns.

For space reasons, we only explain *TemPsy* informally; we refer the reader to the extended version [15] of our previous work [17] for a more complete and formal treatment. A *TemPsy* property includes two main entities: a scope and a pattern; both are denoted by keywords with an intuitive syntax. Properties cannot be nested.

Events used in properties are alphanumeric strings, matching the event names logged in the execution trace on which the properties specified in *TemPsy* are meant to be checked. *TemPsy* properties may contain time distances (both between events and from scope boundaries); time distances are expressed with an integer value, followed by the 'tu' keyword, which represents a generic system time unit (i.e., any denomination of time) as suggested in [30]). Chains of events, used in *order* patterns, are defined as comma-separated list of events, possibly with a time distance between each pair of events (denoted with the '#' symbol).

As an example, the property "Event B shall happen at least 4 time units before the third occurrence of event Y" is expressed in *TemPsy* as "`before 3 Y at least 4 tu eventually B`".

The semantics of patterns in *TemPsy* is defined as follows:

Universality. An event should occur across the entire execution trace; the corresponding keyword is `always`.

Existence. The *existence* pattern can be expressed in four variants, using the following syntax: "`eventually [(at least | at most | exactly) m] A`", where the brackets indicate an optional part and the vertical bar represents an alternative. The basic variant indicates that event A will eventually happen at least once; the other three variants are used to express a bounded existence pattern, which indicates that event A will eventually happen at least/at most/exactly `m` times.

Absence. In addition to stating that a certain event *never* occurs in the given scope, *TemPsy* makes also possible to specify that a specific number of occurrences of the same event should not happen, as in "`never exactly 2 A`", which indicates that A should never occur exactly twice.

Precedence. This pattern indicates the precondition relationship between a pair of events (respectively, the two blocks of a chain). A *block* of an event chain can be either an atomic event or a sequence of individual events with optional constraints on the time distance between two consecutive events within the block. In this pattern, the occurrence of the second event (respectively, block) depends on the occurrence of the first event (respectively, block). Based on this definition, we added support for timing information to enable expressing the time distance between two adjacent events. For example, the pattern "A `preceding at most 5 tu` B, `#at least 2 tu` C" indicates that the event A is the precondition of the block "B followed by C". In this pattern, A (left-hand side of 'preceding') represents the first block, while the expression "B, `#at least 2 tu` C" represents the second block. The time distance between the two blocks, specified right after 'preceding' and equivalent to the distance between the timestamp of the first element of the second block (i.e., B) and the last element of the first block (i.e., A), should be at most 5 (time units). The time distance between a specific pair of consecutive events in the same block (denoted with a # symbol), in this case B and C, should be at least 2.

Response. This pattern specifies the cause-effect relationship between a pair of events (respectively, the two blocks of a chain) in which the occurrence of the first event (respectively, block) leads to the occurrence of the second event (respectively, block). Similarly to *precedence*, we added support for timing information to enable expressing the time distance between two adjacent events.

3 CHARACTERIZATION OF *TEMPSY* VIOLATIONS

At the basis of our model-driven approach for trace diagnostics there is a precise characterization of the violations that can occur with each type of property. Since *TemPsy* does not allow for arbitrary nesting of temporal expressions (i.e., the structure of temporal properties in *TemPsy* is flat), such a characterization yields a finite set of possible violations, which can be uniquely associated with each type of property. In the following, we describe the different *violation types* that characterize *TemPsy* properties. In the examples, we represent an execution trace as a list (denoted with brackets) of trace elements; each trace element is a pair (denoted with parentheses) consisting of the event name and a timestamp.

`UNOC` *UNexpected OCcurrence*. This type of violation is triggered by unexpected occurrences of the event specified in an *absence* or *existence* pattern.

The original version of the *absence* pattern, by definition, is violated by any occurrence of the event specified in the pattern. The variant of the *absence* pattern having `exactly` as comparison operator triggers this violation type when the number of event occurrences in the trace is equals to the number specified in the pattern.

As for the *existence* pattern, the two variants with `at most`/`exactly` in the constraint on the number of occurrences are violated when the bound is exceeded because of an unexpected occurrence. For instance, given the trace $[(A, 2), (A, 3), (A, 5)]$ and the *TemPsy* property "`globally eventually at most 2 A`", the third trace element is unexpected with respect to the *existence* pattern (which in this case sets a bound of at most two occurrences of A) and hence triggers a `UNOC` violation.

`NSOC` *No-Show OCcurrence*. This type of violation is the dual of `UNOC`: it is triggered upon detecting a missing occurrence of the event specified in a *universality* or *existence* pattern.

In the case of a *universality* pattern, an `NSOC` violation is triggered whenever a trace element does not match the event specified in the pattern.

In the case of an *existence* pattern, the basic variant and the ones with `exactly`/`at least` in the constraint on the number of occurrences are violated when the actual number of occurrences of the event is less than the lower bound specified in the property. For example, the *TemPsy* property "`globally eventually at least 2 A`", when checked on the trace $[(A, 2), (B, 3), (B, 5)]$, yields an `NSOC` violation because the number of occurrences of event A is less than two.

`NSOR` *No-Show ORder*. This type of violation is triggered while checking an *order* pattern, when one of the two blocks of events does not occur according to the order defined by the pattern. For instance, the *TemPsy* property "`globally A preceding B`", when checked on the trace $[(b, 2), (a, 3), (c, 5)]$, gives an `NSOR` violation because there is no occurrence of event A that precedes an occurrence of event B.

`WTO` *Wrong Temporal Order*. This type of violation is specific to an *order* pattern that contains a constraint on the time distance between the two blocks of the pattern; it is triggered when this constraint is violated. For example, the trace $[(A, 2), (B, 6), (C, 15)]$ violates the property "`globally A, #at least 3 tu B preceding at most 2 tu C`" and yields a `WTO` because the distance between

the two blocks—i.e., the difference between the timestamp of the first element of the second block (15, for C) and the timestamp of the last element of the first block (6, for B)—is more than the prescribed bound of 2 time units.

WTC *Wrong Temporal Chain.* This type of violation is specific to an *order* pattern that contains at least one constraint on the time distance between two consecutive events in one of the two blocks; it is triggered when such a constraint is violated. For instance, the trace $[(A, 2), (B, 3), (C, 5)]$ violates the property "`globally A, #at least 3 tu B preceding C`" because, though C is preceded by the event chain A, B, the distance between event A and B (3-2=1 time unit in this case) is less than the prescribed lower bound (3 time units); a `WTC` violation is then triggered.

WTOC *Wrong Temporal Order and Chain.* This type of violation is specific to an *order* pattern that contains both a constraint on the distance between the two blocks and at least one constraint on the distance between two consecutive events in one of the two blocks. It is triggered when both types of constraints are violated; in other words, this violation combines (and supersedes) the `WTO` and `WTC` violations when both occur within the same property.
For instance, the trace $[(A, 2), (B, 3), (C, 9)]$ violates the property "`globally A, #at least 3 tu B preceding at most 2 tu C`" and yields a `WTOC` because 1) the distance between events A and B in the first block (1 time unit) violates the constraint "`at least 3 tu`", and 2) the distance between the two blocks (6 time units) violates the constraint "`at most 2 tu`".

4 MODEL-DRIVEN TRACE DIAGNOSTICS

Our technique for trace diagnostics is based on the idea of retrieving, from a trace λ that violates a property ρ, the diagnostic information that is associated with the possible violations admitted by ρ, as determined by the characterization of *TemPsy* violations presented in the previous section. Notice that we expect the trace diagnostics to be run after performing trace checking (for example, by means of TemPsy-Check [18]): this is why *we assume that λ is known to violate ρ.*

The technique follows a model-driven approach, to fulfill requirement R1 stated in Section 1. More specifically, we use the meta-model for traces introduced in our previous work [17]; we define OCL *queries* on this meta-model to analyze violations and collect diagnostic information. The queries are supported by a set of auxiliary OCL functions, which are optimized based on the structure of the targeted *TemPsy* property for achieving better performance when processing large traces, with a huge number of events.

4.1 Overview of the approach

We recall that our meta-model for traces contains a class `Trace`, which is composed of a sequence of `TraceElements`; each `TraceElement` is represented by a pair `(event, timestamp)`, whose elements correspond to the actual event recorded in the trace and the time at which it occurred. One of the attributes of class `Trace` is `properties`, which is a collection of `TemPsyExpressions`, representing the *TemPsy* properties to analyze for trace diagnostics. More details on this meta-model are available in [17].

The approach takes as input a log file (i.e., a trace) and a set of *TemPsy* properties for which to compute the diagnostics; the input

```
1  let property:TemPsy::TemPsyExpression =
2              self.properties->at(POS),
3      subtraces:Sequence(Tuple(begin:Integer,
4                          end:Integer)) =
5              applyScope(property.scope)
6  in  subtraces->iterate(
7   subtrace:Tuple(begin:Integer,end:Integer);
8   result:Sequence(Tuple(begin:Integer, end:
        Integer,violations:OclAny)) = Sequence{} |
9   let newViols:OclAny = diagInf(subtrace.begin,
        subtrace.end, property.pattern) in
10   if newViols->notempty() then
11      result->append(
12        Tuple{begin:Integer = subtrace.begin,
13              end:Integer = subtrace.end,
14              violations:OclAny = newViols})
15   else result endif)
```

Figure 1: The template for OCL queries on a trace for collecting diagnostic information

trace is converted to an instance of class `Trace`, while the properties are converted to instances of the `TempsyExpression` class.

At the core of our approach there is the evaluation of OCL queries on the instance of the `Trace` class, to collect diagnostic information from the trace instance based on the *TemPsy* properties to analyze. These queries follow the template shown in Figure 1: for a given *TemPsy* property provided in input, they apply the semantics of the pattern used in the input property on a set of sub-traces, as defined by the scope used in the property. Through the application of the pattern semantics, the query collects the relevant diagnostic information, specific to the type(s) of violations found. More in detail, the expression at lines 6–15 iterates through each sub-trace to collect diagnostic information. In each iteration, if a violation is found in the sub-trace, a new triple that contains the diagnostic information is collected (and appended to the sequence `result`, see lines 11–14). Each triple consists of two integers (`begin` and `end`), which indicate the boundaries of each trace segment, and the diagnostic information for the violation (`violations`). This information has type `OclAny` since the actual data type varies based on the type of violation that is found.

The template in Figure 1 contains three placeholders, `POS`, `applyScope`, and `diagInf`, which are underlined in the "`let`" clauses. Placeholder `POS` represents a positive integer, used by the attribute `self.properties` (through the function `at`) to access the *TemPsy* property to be analyzed (from the set of input properties). Placeholder `applyScope` (at line 5) represents an auxiliary function that takes the scope used in the property (accessed through the expression `property.scope`) as input and returns a list of sub-traces, each of which is denoted by the positions of the two boundaries, as defined by the scope semantics. Depending on the type of the scope, there are five auxiliary functions to substitute for the placeholder: `applyScopeGlobally`, `applyScopeBefore`, `applyScopeAfter`, `applyScopeBetweenAnd`, and `applyScopeAfterUntil`. The definition of these functions is not included in this paper since it is identical to the one presented in our previous work [17]. Placeholder `diagInf` (at line 9) represents an auxiliary function that takes as

Algorithm 1: diagInfExistence

Input: *begin*, *end*: the boundaries of a sub-trace; *pattern*: an instance of the *existence* pattern of the form "`eventually [op n] E`"
Output: *result*: a pair which contains a violation type and a list of locations related to the violation

```
1  E ← event name in pattern
2  op ← comparison operator of the bound on the number of
     occurrences of event E
3  n ← threshold of the number of occurrences of event E
4  result ← null, locations ← []
5  for i ← begin to end do
6      if self.traceElements[i] = E then
7          locations.append(i)
8  count ← locations.size()
9  if op ≠ null then
10     if count < n && op ≠ "at most" then
11         result ← (NSOC, locations)
12     else if count > n && op ≠ "at least" then
13         result ← (UNOC, locations[n + 1, count])
14 else if count = 0 then result ← (NSOC, [])
15 return result
```

input the two boundaries of a sub-trace and the pattern used in the property (accessed through the expression `property.pattern`) and returns the diagnostic information about the violations found in the sub-trace for the pattern given in input. There are five functions to substitute for this placeholder: `diagInfUniversality`, `diagInfExistence`, `diagInfAbsence`, `diagInfPrecedence`, and `diagInfResponse`. The complete definitions in OCL of these functions are available in the first author's PhD thesis [11, Chapter 4]. For space reasons, in the following subsections we illustrate only two of these functions; for readability and conciseness, all the code snippets presented are written using OCL pseudocode.

4.2 Diagnostics for the "Existence" pattern

The function for collecting diagnostic information for the *existence* pattern is shown in Algorithm 1. The function takes as input the boundaries of a sub-trace and an instance of the *existence* pattern of the form "`eventually [op n] E`", and returns a pair that contains the type of violation and a list of locations with events related to the violation. After reading the parameters *E*, *op*, and *n* from the instance of the *existence* pattern (lines 1–3), the function initializes the variable *result* (storing the output value) to null and the auxiliary variable *locations* to an empty list. The latter is then populated with all the locations in the sub-trace in which event *E* occurs (loop at lines 5–7). Then, depending on the threshold *n* of the bound on the number of occurrences of event *E*, the function determines the violation type and the occurrences that trigger the violation (lines 9–15).

As described in Section 3, when an *existence* pattern is violated, it can yield either a `UNOC` or `NSOC` violation type, depending on the variant of the pattern. If the variant of the *existence* pattern has a comparison operator, then the procedure checks which violation should be triggered. More precisely, if the comparison operator is either "`at least`" or "`exactly`" (line 10) and the number of occurrences of *E* is less than *n*, then the violation type will be `NSOC` and the list of locations will contain all the locations in which event *E* occurs (line 11); the latter are selected because they represent the "fragment" in the sub-trace in which the missing occurrence(s) of *E* was supposed to appear. Otherwise, if the number of occurrences of event *E* is more than *n* and the comparison operator is either "`at most`" or "`exactly`" (line 12), then the violation type will be `UNOC` and the list of locations will contain all the locations with the "extra" occurrences of event *E*, i.e., all the locations after the *n*-th occurrence of event *E* (line 13). If the *existence* pattern is in its basic variant (i.e., with no explicit constraint on the number of occurrences), if there is no occurrence of event *E* in the sub-trace, then the violation type is set to `NSOC`, with an empty list of locations related to the violation (line 14). The function ends by returning the value of variable *result*.

4.3 Diagnostics for the "Precedence" pattern

Function `diagInfPrecedence` (Algorithm 2) defines the algorithm for the variant of the *precedence* pattern that contains no time constraint on the distance between the two blocks. The function takes as input the two boundaries of a sub-trace and an instance of the *precedence* pattern of the form "$block_1$ `preceding` $block_2$". We recall that both blocks of the pattern can be either an atomic event or a chain of events with optional constraints on the time distance between two consecutive events within the block. The function returns a list of triples, each of which contains a violation type, the location of the occurrence of $block_2$ related to the violation, and the location of the corresponding occurrence of $block_1$.

After reading $block_1$ and $block_2$ from the instance of the *precedence* pattern (line 1), the function initializes (lines 2–6) variable *result* (storing the return value) to an empty list and some auxiliary variables: $size_1$ and $size_2$ store, respectively, the size of $block_1$ and of $block_2$; *firstOfBlock1* and *firstOfBlock2* contain the first event defined, respectively, in $block_1$ and $block_2$; the tuple (i_1, t_1) and (i_2, t_2) are used to track whether the trace element being matched is part of an occurrence of $block_1$ (respectively, $block_2$): the first element of the tuple stores the position within the block of the next event to be matched while the second tuple element stores the timestamp of the previous trace element matched at position $block_1[i_1 - 1]$ and $block_2[i_2 - 1]$, respectively; variable *flag* is used to track whether the ongoing match of $block_1$ is consistent with the distance constraints (if defined) within the block; variable l_w contains the location of the last occurrence of $block_1$ that was found to violate the time constraints within the block.

As presented in Section 3, when a *precedence* pattern with no time constraint on the distance between the two blocks is violated, it can yield either a `NSOR` or a `WTC` violation type; the latter case can occur only if $block_1$ contains a distance constraint[1]. The body of function `diagInfPrecedence` is mainly constituted by a loop that traverses the input sub-trace, trying to match each trace ele-

[1]Based on the semantics of the *precedence* pattern [11], if $block_2$ contains a distance constraint that is violated it will not be matched, i.e., it will not be considered an "effect" that must be preceded by a "cause" (represented by $block_1$).

Algorithm 2: diagInfPrecedence

Input: *begin*, *end*: the boundaries of a sub-trace; *pattern*: an instance of the *precedence* pattern of the form "$block_1$ `preceding` $block_2$"

Output: *result*: a list of triples, each of which consists of a violation type, the location of the occurrence of $block_2$ related to the violation, and the location of the corresponding occurrence of $block_1$

```
1  block1, block2 ← read the blocks from pattern
2  result ← []
3  size1, size2 ← the sizes of block1 and block2
4  firstOfBlock1 ← block1.first().event, firstOfBlock2 ←
     block2.first().event
5  (i1, t1) ← (1, 0), (i2, t2) ← (1, 0)
6  flag ← true, lw ← 0
7  for i ← begin to end do
8      elem ← self.traceElements[i],
         (e, t) ← (elem.event, elem.timestamp)
9      if e = firstOfBlock1 then (i1, t1, flag) ← (2, t, true)
10     else if i1 > 1 then (i1, t1, flag) ←
         matchSecondaryBlock(block1, (i1, t1, flag), (e, t))
11     if lw > 0 && ! flag then (i1, t1) ← (1, 0)
12     else if i1 = size1 + 1 then
13         if flag then break
14         else
15             lw ← i
16             (i1, t1) ← (1, 0), (i2, t2) ← (1, 0)
17     if e = firstOfBlock2 then (i2, t2) ← (2, t)
18     else if i2 > 1 then (i2, t2) ← match(block2, i2, t2, (e, t))
19     if i2 = size2 + 1 then
20         if lw = 0 then v ← (NSOR, i, null)
21         else v ← (WTC, i, lw)
22         result.append(v)
23         (i1, t1) ← (1, 0), (i2, t2) ← (1, 0)
24 return result
```

ment between *begin* and *end* with $block_1[i_1]$ (lines 9–16) and with $block_2[i_2]$ (lines 17–23). Two local variables *e* and *t* are used in each iteration to store the event name and timestamp of the current trace element (line 8).

If the trace element is a match for the first event of $block_1$ (line 9), the function sets the position i_1 to 2, assigns the timestamp of the trace element to t_1, and resets the *flag* to `true`. Otherwise, if variable i_1 is greater than 1, the function invokes the auxiliary function `matchSecondaryBlock` to check whether the current trace element is part of $block_1$ (line 10). Function `matchSecondaryBlock` (not shown here for space reasons), when invoked with input $(block_1, (i_1, t_1, flag), (e, t))$ returns the tuple $(i_1 + 1, t, flag$ && `true`$)$ if the trace element is a valid match for $block_1[i_1]$; it returns the tuple $(i_1 + 1, t, flag$ && `false`$)$ if the trace element matches the event defined at $block_1[i_1]$ but violates the constraint on the distance between $block_1[i_1 - 1]$ and $block_1[i_1]$; it returns the tuple $(1, 0, flag)$ otherwise. At line 11, the function continues to check whether the matched event is part of an invalid occurrence of $block_1$. If there is already an invalid occurrence of $block_1$ (i.e., $l_w > 0$) and the matched event violates the constraint on the distance between $block_1[i_1 - 1]$ and $block_1[i_1]$, the algorithm resets the tuple (i_1, t_1) (line 11). If it is not the case and $block_1$ is fully matched, if variable $flag_1$ is `true`, the function stops the collection of diagnostic information (line 13); otherwise, the variable l_w is updated with the position *i* of the current *invalid* occurrence of $block_1$ (line 15) and the variables i_1, t_1, i_2, t_2 are reset (line 16).

As an example, given the trace $[(A, 2), (B, 6), (C, 15)]$, $begin = 1$, $end = 3$, and (a property with) the precedence pattern "A, `#at least 3 tu` B `preceding` C", the function will execute the following main steps: 1) when *i* is 1, $(i_1, t_1, flag)$ is set to $(2, 2, true)$ (line 9), since the first trace element matches the first event of $block_1$; 2) when *i* is 2, $(i_1, t_1, flag)$ is set to $(3, 6, true)$ (line 10) and breaks the loop at line 13, since the function finds a match of $block_1$. Notice that the rest of the iteration does not impact the matching of $block_1$.

If the function has not yet found a valid occurrence of $block_1$, in the remainder of the iteration, it checks whether the current trace element is part of an occurrence of $block_2$ (lines 17–23). If an occurrence of the first event of $block_2$ is detected, the variable i_2 is set to 2 and variable t_2 is set to the timestamp of current trace element (line 17). Otherwise, if variable i_2 is already greater than 1, the algorithm calls function `match` to match the current trace element with $block_2[i_2]$ (line 18).

The algorithm reports a violation (lines 19–23) when $block_2$ is completely matched. More specifically, if an invalid occurrence of $block_1$ has not yet been found (i.e., l_w is still 0), it means that the pattern is violated because the occurrence of the second block is not matched by an occurrence of the first block; hence, an `NSOR` violation type is reported, together with the location of the current trace element (indicating the occurrence of $block_2$ just matched) and the `null` value indicating there is no corresponding occurrence of $block_1$ (line 20). Otherwise, a `WTC` violation type is reported, with the location of the current trace element (indicating the occurrence of $block_2$ just matched), and the location l_w of the corresponding invalid occurrence of $block_1$ (line 21). The diagnostic information is then added to variable *result* (line 22) and the tuples (i_1, t_1) and (i_2, t_2) are reset (line 23). After analyzing the entire sub-trace, the function ends by returning variable *result* (line 24).

5 TOOL SUPPORT

We have implemented our model-driven procedure for trace diagnostics in the tool TemPsy-Report, publicly available [12, 13]. This tool extends the implementation of our tool TemPsy-Check [18] and is also based on Eclipse OCL [20]. The tool works as follows: given a trace and a set of *TemPsy* properties (previously determined, by TemPsy-Check, as violated on the input trace), TemPsy-Report builds OCL queries following the template shown in Figure 1, depending on the type of the scope and pattern used in each *TemPsy* property. We have implemented all the OCL functions (defined on class `Trace`) for *TemPsy* scopes and patterns as described in the previous subsection, to collect diagnostic information. The evaluation of the OCL queries is done through the `evaluate` function

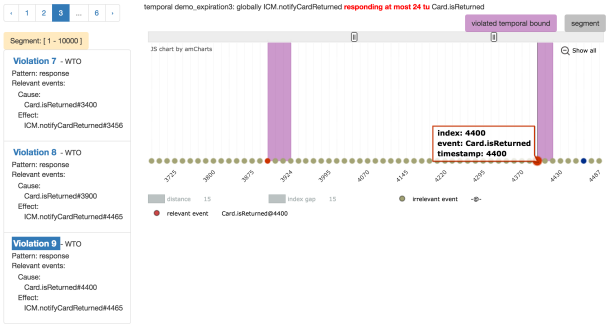


Figure 2: Screenshot of our visualization tool for *TemPsy* trace diagnostics

provided by Eclipse OCL. TemPsy-Report produces a textual version of the diagnostic information and also saves the latter in a document-based MongoDB [28] database, for further processing.

The textual version of the diagnostic information is quite complex and can be cumbersome to inspect, especially when a violation can be triggered by different causes. For this reasons, we have developed an interactive visualization tool for trace diagnostic information, publicly available [14]. The tool is implemented as a Web application using JavaScript Charts [1], Meteor.js [25], AngularJS [24], and ElasticSearch [21].

The visualization tool loads the diagnostic information saved by TemPsy-Report in the MongoDB database. It displays the trace using a chart component; a data point in the chart corresponds to a trace element. The tool UI includes several features to provide an easy and useful diagnostics session: showing the details of a trace element when hovering over its data point; zooming the trace in and out with an adaptive granularity of the data points; navigating the trace horizontally; navigating the list of violations; marking the occurrences of events related to a violation; highlighting the reason of a violation. Some of these features are shown in the screenshot in Figure 2: on the left side, there is a navigable list of the violations found in the trace; the red callout on the right side shows the details (e.g., the timestamp) for an event related to the violation; in the top part, the red marking in the property text highlights the constraint that was violated.

6 EVALUATION

As stated by requirement R2 in Section 1, our solution for model-driven trace diagnostics is expected to support very large traces, with millions of events, such that violations in the trace could be processed within seconds. To check the fulfillment of this requirement, we evaluated the scalability of TemPsy-Report by investigating the relation between the execution time and some structural properties of a trace, such as the length and the number of violations. More specifically, we consider the following research questions:

RQ1) What is the relation between the execution time of TemPsy-Report and the length (i.e., number of logged events) of a trace?

RQ2) What is the relation between the execution time of TemPsy-Report and the number of violations (with respect to a given TemPsy property) contained in a trace?

Table 1: *TemPsy* properties used for the evaluation

P1: globally always A
P2: globally never B
P3: globally eventually at least 2 A
P4: globally eventually at most 3 A
P5: globally A responding at most 1000 tu B
P6: globally A responding exactly 1000 tu B
P7: globally A preceding at most 6000 tu B
P8: globally A preceding at least 100 tu B
P9: globally A preceding exactly 100 tu B
P10: globally A, B preceding at least 1000 tu C, D
P11: globally A responding at least 1000 tu B, C
P12: globally A responding B

6.1 Benchmark and Settings

Benchmark. The benchmark for evaluating TemPsy-Report is constituted by a set of *TemPsy* properties and by a set of traces.

As for the properties, we employed the ones already used in our previous work [17], extracted from the requirements specification documents of an eGovernment application developed by our partner. Since we focus on the scalability of the trace diagnostics procedure, we considered the 12 (out of 47) properties that use the *globally* scope: the semantics of this scope guarantees that a pattern is analyzed through the entire length of the trace. These properties are shown in sanitized form in Table 1; for confidentiality reasons, we only keep the structure of each property, in terms of *scope* + *pattern*, and denote events with uppercase letters.

Regarding the traces, as already done in [17], we used *synthesized* traces, to cover a large spectrum of lengths while having a great, controlled diversity in terms of occurrences of violations in the traces. More specifically, by using synthesized traces we were able to control in a systematic way the factors (i.e., trace length, number and type of violations) required to answer the research questions, while setting other factors (e.g., distance between events) randomly, to avoid any bias.

We synthesized these traces using a trace generator program, which we also implemented. This program takes in input a *TemPsy* property and configuration options, and generates traces that violate the input property. The generator avoids bias by distributing the (events leading to) violations evenly, assigning them random positions within the slots determined by the input parameters, such as the trace length and the number of violations. The position and the order of the events related to a violation are generated randomly by taking into account the temporal and timing constraints prescribed by the semantics of the pattern used in the input property. Positions in the trace that are not related to the property are filled with a dummy, irrelevant event. In the following we briefly describe the trace generation strategy for each pattern; next to each pattern name, we also indicate the corresponding properties from Table 1.

Universality (P1). There is only one possible violation type for this type of pattern: `NSOC`. The generator first randomly generates the violation positions and then inserts in them a dummy event; all the other positions will have the event indicated in the property.

Existence (P3, P4). The trace generation strategy depends on the bound indicated in the property, in terms of comparison operator and bound value m.

If the bound is expressed as "`at least` m" (as in property P3), the violations will be of type `NSOC`; the number of their occurrences (i.e.,

the number of occurrences of the event indicated in the property) is set to the minimum between $m - 1$ and the value of the input parameter "number of violations". If the bound is expressed as "`at most` m" (as in property P4), the violations will be of type `UNOC`. The number of the occurrences of the event indicated in the property is set to the maximum between $m + 1$ and the value of the input parameter "number of violations". In both cases, the position of these events is set randomly; all other positions in the trace are filled with a dummy event. A similar process is followed for the other variants of the pattern, not used in our benchmark.

Absence (P2). This pattern can only be violated by triggering `UNOC` violations. The generator first randomly generates the violation positions and then inserts in them the event indicated in the property (e.g., B in the case of P2); all the other positions will be filled with a dummy event.

Precedence (P7–P10). For this type of properties, there may be more than one type of violations, depending on the structure of the pattern; to avoid any bias, the generator takes as input also the violation type to be generated and produces a set of traces containing only the required violation type. For example, all properties P7–P10 can lead either to `NSOR` or to `WTO` violations.
Given the number of violations n as parameter, the generator first divides the trace into n segments with the same length, and then randomly inserts a specific violation into a position within each segment, taking into account the distance constraints prescribed by the pattern. For example, to generate n `NSOR` violations, the generator produces n occurrences of the second block (e.g., the event chain "C, D" for property P10), without inserting the matching occurrences of the first block. To generate n `WTO` violations, the generator produces n pairs of the two blocks indicated in the property, making sure that their positions in the trace violate the constraint on the time distance between them. The strategy for choosing the value of the distance between the two blocks depends on the type of the comparison operator used in the distance constraint. If the bound is expressed as "`at most` m" (as in property P7), the distance is randomly generated using a uniform distribution on the range $[m + 1, m * 1.1]$; if the bound is expressed as "`at least` m" (as in properties P8 and P10), the distance is randomly generated using a uniform distribution on the range $[1, m - 1]$; if the bound is expressed as "`exactly` m" (as in property P9), the distance is randomly generated using a uniform distribution on the range $[1, \ldots, m - 1, m + 1, \ldots, m * 1.1]$. A similar strategy is used to generate violations of type `WTC` and `WTOC`.

Response (P5–P6, P11–P12). The trace generation for this pattern is similar to the one described above for the *precedence* pattern, taking into account that *response* is the dual of *precedence*. Also in this case, the generator takes as input the violation type to be generated and produces a set of traces that contains only the required violation type. For example, properties P5, P6, and P11 can lead either to `NSOR` or to `WTO` violations, while P12 can lead only to an `NSOR` violation.

Settings. The results reported in this section have been measured (by invoking the `System.currentTimeMillis()` method of the standard Java library) on a desktop computer with a 3 GHz Intel Dual-Core i7 CPU and 16 GB of memory, running Eclipse DSL Tools v. 4.6.0M3 (Neon Milestone 3), JavaSE-1.7 (Java SE v. 1.8.0_25-b17, Java HotSpot (TM) 64-Bit Server VM v. 25.25-b02, mixed mode), and Eclipse OCL v. 6.0.1. All measurements reported correspond to the average value over 100 runs of the trace diagnostics procedure (on the same trace, for the same property).

6.2 Scalability Analysis

Methodology. To answer RQ1, for each property (and for each type of violation, `NSOR` and `WTO`, in the case of properties P5–P11), we generated ten traces with various lengths from 100K to 1M, with a 100K step increment; in the trace generator program we fixed the number of violations[2] to 1000. To answer RQ2, for each property but P3[3] (and for each type of violation, `NSOR` and `WTO`, in the case of properties P5–P11), we generated ten traces varying the number of violations[2] from 1K to 10K, with a 1K step increment; in the trace generator program we fixed the length of the trace to 1M. In both cases, we ran TemPsy-Report to analyze, for each property, the ten corresponding traces.

Results. The plots in Figure 3 show the relation between the execution time of TemPsy-Report and the trace length (RQ1). The execution time of TemPsy-Report on a trace containing `NSOR` violations is denoted by adding a superscript † to the property name, while a superscript ‡ is used to indicate the execution time on a trace with `WTO` violations. We split the plots into two parts (Figures 3a and 3b), to better highlight the execution time for trace diagnostics of the properties with a *precedence* pattern with a distance constraint of the form "`at least` n `tu`" (P8 and P10).

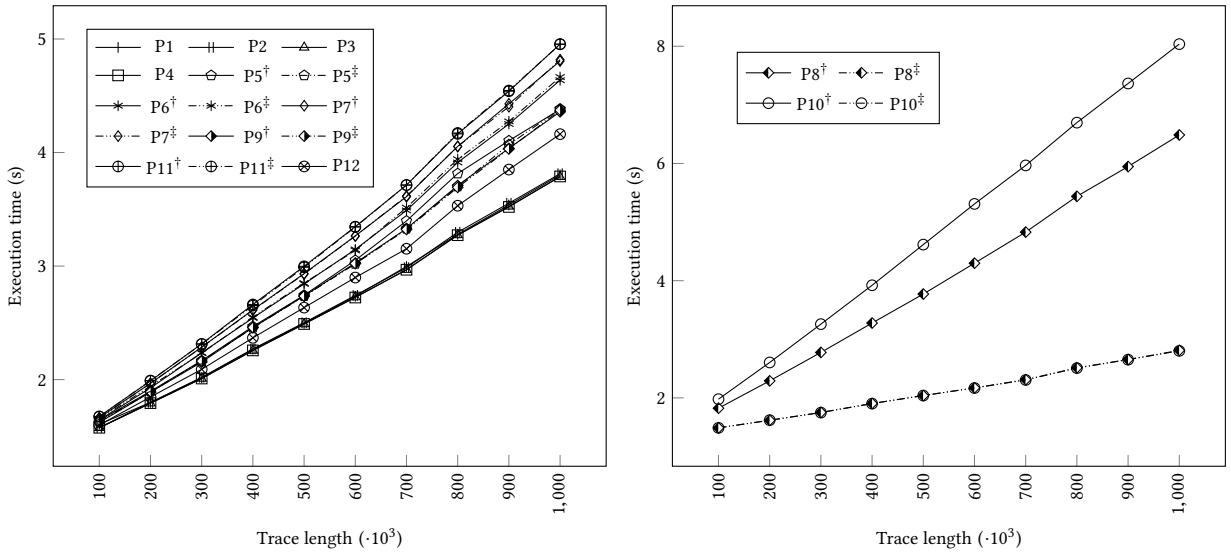
The answer to RQ1 is that the TemPsy-Report tool scales linearly with respect to the trace length; the execution time ranges from about 1.5 s to 8.2 s, depending on the pattern used in the property and the violation type in the trace.

The plots in Figure 4 show the relation between the execution time of TemPsy-Report and the number of violations (RQ2) contained in a trace. We use the same notation as above to distinguish between `NSOR` and `WTO` violations. Also in this case, the plots for the diagnostics of properties P8 and P10 are separated.

The answer to RQ2 is that the number of violations contained in a trace makes no tangible impact on the execution time of the TemPsy-Report tool, which stays approximately constant as the number of violations increases; the execution time ranges from about 3.8 s to 8.2 s depending on the pattern used in the property and the violation type in the trace.

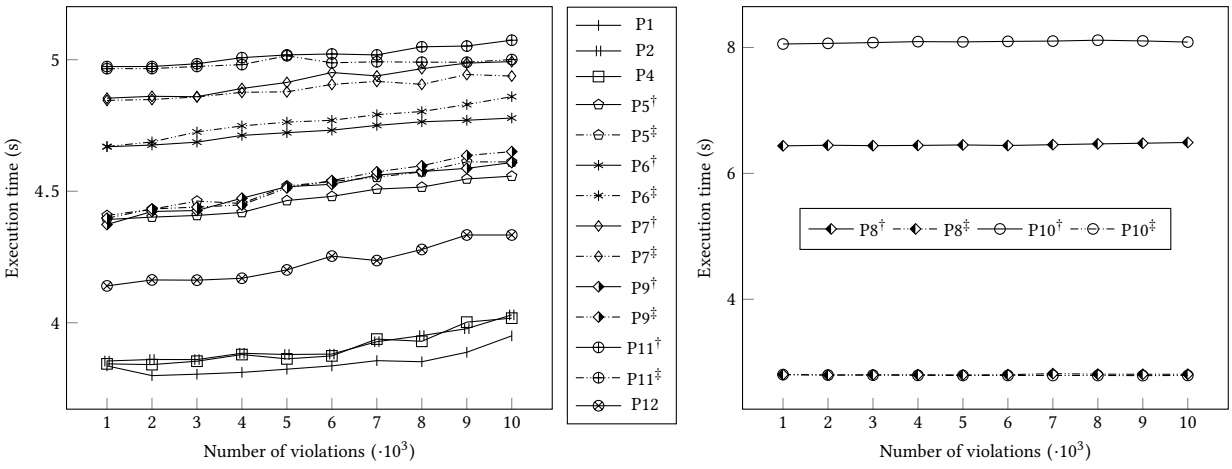
We also gain additional insight by inspecting the plots in Figures 3 and 4: TemPsy-Report takes less time to collect diagnostic information for the properties using the *universality*, *existence*, and *absence* patterns (e.g., P1–P4) than for the properties using the *precedence* and *response* patterns. Also, Figures 3b and 4b show that the tool takes less time for analyzing traces with `WTO` violations than traces with `NSOR` violations, when considering properties with a *precedence* pattern with a distance constraint of the form "`at least` n `tu`" (P8 and P10). This is due to the implementation of the corresponding OCL function, which—in the worst case—has

[2] In the case of the *universality*, *absence*, and *existence* patterns, the number of violations actually represents the number of occurrences of a specific event *that lead* to a violation.
[3] Property P3 was not used to answer RQ2 because varying the number of violations does not make sense from a scalability analysis standpoint: the property is violated only when there is no occurrence or just one occurrence of event A.



(a) P1–P7, P9, P11–P12 (b) P8, P10

Figure 3: Execution time of TemPsy-Report for collecting diagnostic information from faulty traces (number of violations fixed to 1000, various lengths)



(a) P1–P7, P9, P11–P12 (b) P8, P10

Figure 4: Execution time of TemPsy-Report for collecting diagnostic information from faulty traces (various numbers of violations, trace length fixed to 1M)

to check the order of events blocks, the distance between them, and also the distance between the individual events within a block. Finally, the results also show that TemPsy-Report takes longer for collecting diagnostic information for an *order* pattern in which the blocks are event chains rather than single event (as in the case of P10 and P11); again, this is due to the implementation, which has to check for all events matching a single block and the corresponding time constraint between the individual events in the block.

Threats to validity. The main threat to validity to the results presented above is the intrinsic presence of errors in TemPsy-Report. We tried to compensate for this by thoroughly testing the tool with traces and properties for which the oracle (in terms of diagnostic information) was previously known. Another potential threat is the fact that we have performed trace checking on *synthesized* traces. Real execution traces might be different, in terms of violation types occurring in them, due to a different distribution of event occur-

rences along the traces, both in terms of order and time distance. However, as explained at the beginning of this section, for the purpose of scalability analysis synthesized traces are better than real ones as they guarantee we have the data to perform our analysis by controlling certain factors and varying others randomly. Another threat is given by the use of Eclipse OCL; one could get different results by using another OCL tool, with lower performance. We chose Eclipse OCL for its scalability (see [36]).

Discussion. The evaluation results presented above show the feasibility of applying our model-driven approach for trace diagnostics, in realistic settings, of temporal properties expressed in *TemPsy*. The tool scales linearly with respect to the length of the input trace and keeps approximately constant performance with respect to the number of violations contained in the trace.

The performance of our TemPsy-Report tool, which can analyze very large traces (with one million events) in less than ten seconds, makes it a viable technology for adoption in contexts where MDE is the mainstream practice within the software development process.

7 RELATED WORK

Although run-time verification (and trace checking) has been a very active area of research in the last 15 years [4, 31], little or no attention has been paid to the problem of trace diagnostics. The work closest to our approach is the one by Ferrère et al. [23], which proposes an error diagnostics algorithm for trace diagnostics of Signal Temporal Logic (STL) formulae over a continuous signal; the algorithm computes temporal implicants, i.e., small subsignals that are sufficient to imply violations. This algorithm has been recently integrated into the AMT 2.0 tool [35], which supports qualitative and quantitative analysis of hybrid continuous signals. The main difference with our approach is that AMT does not provide a model-driven approach, which is one of the main requirements (see Section 1) set by the context in which we have developed TemPsy-Report; furthermore, AMT focuses on signal-based, continuous time applications with STL specifications while TemPsy-Report considers event-based, discrete time traces and a restricted, pattern-based temporal specification language.

We have also surveyed the trace diagnostics support in the tools that were contestants of the "offline monitoring" track of the 2014 and 2015 international Competition on Software for Runtime Verification (CSRV 2014 [3] and CSRV 2015 [22]). Four of the tools (STePr, AgMon [29], LogFire [27], OptySim [8]) are not publicly available; another tool RiTHM-v2.0 [34], is available but does not work when executed by following the instructions specified in the *README* file on its GitHub page [39]; the remaining tools that we analyzed are: Breach [10], MonPoly [5], QEA [37], SOLOIST+ZOT [7], and RV-Monitor [33]. Table 2 summarizes our findings, indicating whether a tool produces a boolean output (column "boolean"), whether it takes into account all the (events leading to) violations in a trace (column "positions"), whether it provides detailed diagnostic information, such as the cause of a violation (column "cause"), and whether it is based on MDE technologies (column "MDE"). Although all tools yield a boolean result, only few provides additional information. Breach, which can check digitized traces against Signal Temporal Logic (STL) specifications, provides a graphical visualization of the violations in the trace. MonPoly prints out the last log entries read before finding a violation. QEA stops checking the trace after the first violation is found and prints the last log entry read before finding the violation. None of these tools provide additional diagnostic information to understand the cause of a violation. RV-Monitor does not report the position of the violations in a trace but it allows for manually writing a violation handler to print some user-(pre)defined diagnostic information. As seen in Table 2, only AMT and TemPsy-Report provide detailed diagnostic information to investigate faulty traces; however, only TemPsy-Report is a tool based on standard MDE technologies such as OCL.

Table 2: Comparison of trace diagnostics features among trace checking/run-time verification tools

Tool	boolean	positions	cause	MDE
Breach [10]	+	+	-	-
MonPoly [5]	+	+	-	-
QEA [37]	+	+/-	-	-
SOLOIST+ZOT [7]	+	-	-	-
RV-Monitor [33]	+	-	+/-	-
AMT [35]	+	+	+	-
TemPsy-Report	+	+	+	+

Trace diagnostics is an activity related to *falsification* [9], e.g., finding counterexamples to a system specification, and to the analysis of counterexamples in the context of model checking [6, 38].

Finally, trace diagnostics can be seen as a particular case of query on a trace of events, as typically done in Complex Event Processing (CEP [32]) settings; strategies for the synergetic integration of RV and CEP are discussed by Hallé [26].

8 CONCLUSIONS

In this paper we described a model-driven approach for the diagnostics of the verdicts yielded after checking a property over a trace. We implemented our approach in TemPsy-Report, a tool that provides detailed trace diagnostic information for pattern-based temporal properties expressed in the *TemPsy* language; this information can be displayed in an interactive visualization tool.

The evaluation of TemPsy-Report shows that it can collect diagnostic information from large traces (with one million events) in less than ten seconds. TemPsy-Report scales linearly with respect to the length of the trace and is not affected by the the number of violations contained in the trace.

As part of future work, we plan to conduct a user study to assess how TemPsy-Report can support developers while performing fault localization, by providing useful diagnostic information.

ACKNOWLEDGMENTS

This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), from the Luxembourg National Research Fund (FNR) (grant No FNR/P10/03), and from the University of Luxembourg (grant "MOVIDA").

REFERENCES

[1] amCharts. 2018. JavaScript Charts. Retrieved July 18, 2018 from https://www.amcharts.com

[2] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. 2015. Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. *IEEE Trans. Softw. Eng.* 41, 7 (2015), 620–638.

[3] Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. 2014. First International Competition on Software for Runtime Verification. In *Proc. RV 2014*. LNCS, Vol. 8734. Springer, Heidelberg, Germany, 1–9.

[4] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*. LNCS, Vol. 10457. Springer, Cham, Switzerland, 1–33.

[5] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. 2012. MONPOLY: Monitoring Usage-Control Policies. In *Proc. RV 2011 (LNCS)*, Vol. 7186. Springer-Verlag, Heidelberg, Germany, 360–364.

[6] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. 2012. Explaining Counterexamples Using Causality. *Form. Methods Syst. Des.* 40, 1 (Feb. 2012), 20–40.

[7] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. 2014. SMT-based Checking of SOLOIST over Sparse Traces. In *Proc. FASE 2014 (LNCS)*, Vol. 8411. Springer-Verlag, Heidelberg, Germany, 276–290.

[8] Almudena Díaz, Pedro Merino, and Alberto Salmerón. 2011. Obtaining Models for Realistic Mobile Network Simulations using Real Traces. *IEEE Communications Letters* 15, 7 (July 2011), 782–784.

[9] Ram Das Diwakaran, Sriram Sankaranarayanan, and Ashutosh Trivedi. 2017. Analyzing Neighborhoods of Falsifying Traces in Cyber-physical Systems. In *Proc. ICCPS '17*. ACM, New York, NY, USA, 109–119.

[10] Alexandre Donzé. 2010. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Proc. CAV 2010 (LNCS)*, Vol. 6174. Springer, Heidelberg, Germany, 167–170.

[11] Wei Dou. 2016. *A Model-Driven Approach to Offline Trace Checking of Temporal Properties*. Ph.D. Dissertation. University of Luxembourg. http://hdl.handle.net/10993/29184

[12] Wei Dou. 2018. TemPsy-Report. https://doi.org/10.6084/m9.figshare.6797171.v8

[13] Wei Dou. 2018. TemPsy Report tool. Retrieved July 18, 2018 from https://weidou.github.io/TemPsy-Report/

[14] Wei Dou. 2018. TemPsy Violation Visualization tool. Retrieved July 18, 2018 from http://weidou.github.io/TemPsy-Violation-Visualization

[15] Wei Dou, Domenico Bianculli, and Lionel Briand. 2014. *A Model-based Approach to Offline Trace Checking of Temporal Properties with OCL*. Technical Report TR-SnT-2014-5. SnT Centre - University of Luxembourg. http://hdl.handle.net/10993/16112

[16] Wei Dou, Domenico Bianculli, and Lionel Briand. 2014. Revisiting Model-driven Engineering for Run-time Verification of Business Processes. In *Proc. SAM 2014 (LNCS)*, Vol. 8769. Springer, Cham, Switzerland, 190–197.

[17] Wei Dou, Domenico Bianculli, and Lionel Briand. 2017. A Model-Driven Approach to Trace Checking of Pattern-based Temporal Properties. In *Proc. MODELS 2017*. IEEE Computer Society, Los Alamitos, CA, 323–333.

[18] Wei Dou, Domenico Bianculli, and Lionel Briand. 2017. TemPsy-Check: a Tool for Model-driven Trace Checking of Pattern-based Temporal Properties. In *Proc. International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES 2017 (Kalpa Publications in Computing)*, Vol. 3. EasyChair, Manchester, United Kingdom, 64–70.

[19] Matthew B Dwyer, George S Avrunin, and James C Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proc. ICSE 1999*. ACM, New York, NY, USA, 411–420.

[20] Eclipse. 2018. Eclipse OCL Tools. Retrieved July 18, 2018 from http://www.eclipse.org/modeling/mdt/?project=ocl

[21] Elastic. 2018. ElasticSearch. Retrieved July 18, 2018 from https://www.elastic.co

[22] Yliès Falcone, Dejan Ničković, Giles Reger, and Daniel Thoma. 2015. Second International Competition on Runtime Verification. In *Proc. RV 2015 (LNCS)*, Vol. 9333. Springer, Heidelberg, Germany, 405–422.

[23] Thomas Ferrère, Oded Maler, and Dejan Ničković. 2015. Trace Diagnostics Using Temporal Implicants. In *Proc. ATVA 2015 (LNCS)*, Vol. 9364. Springer International Publishing, Cham, 241–258.

[24] Google. 2018. AngularJS. Retrieved July 18, 2018 from https://www.angularjs.org

[25] Meteor Development Group. 2018. Meter.js. Retrieved July 18, 2018 from https://www.meteor.com

[26] Sylvain Hallé. 2016. When RV Meets CEP. In *Proceedings of RV 2016 (LNCS)*, Vol. 10012. Springer International Publishing, Cham, 68–91.

[27] Klaus Havelund. 2015. Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Transf.* 17, 2 (2015), 143–170.

[28] MongoDB Inc. 2018. MongoDB. Retrieved July 18, 2018 from https://www.mongodb.com

[29] Aaron Kane, Thomas Fuhrman, and Philip Koopman. 2014. Monitor based oracles for cyber-physical system testing: practical experience report. In *Proc. DSN 2014*. IEEE Computer Society, Los Alamitos, CA, 148–155.

[30] Sascha Konrad and Betty H. C. Cheng. 2005. Real-time specification patterns. In *Proc. ICSE '05*. ACM, New York, NY, USA, 372–381.

[31] Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *Journal of Logic and Algebraic Programming* 78, 5 (May/June 2009), 293–303.

[32] David Luckham. 2008. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. In *Proceedings of RuleML '08*. Springer-Verlag, Heidelberg, Germany, 3–3.

[33] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In *Proc. RV 2014 (LNCS)*, Vol. 8734. Springer, Cham, Switzerland, 285–300.

[34] Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2013. RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs. In *Proc. ESEC/FSE 2013*. ACM, New York, NY, USA, 603–606.

[35] Dejan Ničković, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. 2018. AMT 2.0: Qualitative and Quantitative Trace Analysis with Extended Signal Temporal Logic. In *Proc. TACAS 2018 (LNCS)*, Vol. 10806. Springer International Publishing, Cham, 303–319.

[36] István Raáth and Edward Willink. 2012. Fast, Faster and Superfast queries. http://www.eclipse.org/modeling/mdt/ocl/docs/publications/EclipseConEurope2012/FastQueries.pdf. EclipseCon Europe 2012 presentation.

[37] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA. In *Proc. TACAS 2015 (LNCS)*, Vol. 9035. Springer, Heidelberg, Germany, 596–610.

[38] Viktor Schuppan and Armin Biere. 2005. Shortest Counterexamples for Symbolic Model Checking of LTL with Past. In *Proc. TACAS'05 (LNCS)*, Vol. 3440. Springer-Verlag, Berlin, Heidelberg, 493–509.

[39] Yogi Joshi. 2016. RiTHM-v2.0. Retrieved July 18, 2018 from https://github.com/yogirjoshi/maven-repo