A Model-Driven Approach to Trace Checking of Pattern-based Temporal Properties

Wei Dou
SnT - University of Luxembourg
Luxembourg, Luxembourg
dou@svv.lu

Domenico Bianculli
SnT - University of Luxembourg
Luxembourg, Luxembourg
domenico.bianculli@uni.lu

Lionel Briand
SnT - University of Luxembourg
Luxembourg, Luxembourg
lionel.briand@uni.lu

***Abstract*—Trace checking is a procedure for evaluating requirements over a log of events produced by a system. This paper deals with the problem of performing trace checking of temporal properties expressed in *TemPsy*, a pattern-based specification language. The goal of the paper is to present a scalable and practical solution for trace checking, which can be used in contexts where relying on model-driven engineering standards and tools for property checking is a fundamental prerequisite.**

The main contributions of the paper are: a model-driven trace checking procedure, which relies on the efficient mapping of temporal requirements written in *TemPsy* into OCL constraints on a meta-model of execution traces; the implementation of this trace checking procedure in the TEMPSY-CHECK tool; the evaluation of the scalability of TEMPSY-CHECK, applied to the verification of real properties derived from a case study of our industrial partner, including a comparison with a state-of-the-art alternative technology based on temporal logic. The results of the evaluation show the feasibility of applying our model-driven approach for trace checking in realistic settings: TEMPSY-CHECK scales linearly with respect to the length of the input trace and can analyze traces with one million events in about two seconds.

I. INTRODUCTION

Trace checking, also called *trace validation* [1] or *history checking* [2], is a technique for evaluating requirements over a log of recorded events produced by a system. This technique complements verification activities performed before the deployment of a system (e.g., testing and model checking) or during the system's execution (e.g., run-time monitoring).

As part of a larger research collaborative project that we are running with our public service partner CTIE (Centre des technologies de l'information de l'Etat, the Luxembourg national center for information technology), on model-driven run-time verification of business processes [3], we are investigating the use of trace checking for detecting anomalous behaviors of eGovernment business processes and for checking whether third-parties (e.g., other administrations, suppliers) involved in the execution of the process fulfill their guarantees.

The effective application of trace checking goes through two steps: 1) precisely specifying the requirements to check over a trace; 2) defining a procedure for checking the conformance of a trace with respect to the requirements.

Regarding the specification of the requirements to check, many of the existing approaches support some types of temporal properties, usually expressed in some temporal logic, either the classic LTL or CTL, or more complex versions like MFOTL [4] and SOLOIST [5]. However, these specification approaches require strong theoretical and mathematical background, which are rarely found among practitioners. To address this issue, in previous work [6] we proposed *OCLR*, a domain-specific language for the specification of temporal properties, based on the catalogue of property specification patterns defined by Dwyer et al. [7], and extended with additional constructs. The language has been defined in collaboration with the CTIE analysts, based on the analysis of the requirements specifications of an industrial case study. The most recent version of the language, now called *TemPsy* (Temporal Properties made easy) [8], sports a syntax close to natural language, has all the constructs required to express the property specification patterns found in our case study, and has a precise semantics expressed in terms of linear temporal traces. By design, *TemPsy* does not aim at being as expressive as a full-fledged temporal logic. Instead, its goal is to make as easy as possible the specification of the temporal requirements of business processes, by supporting only the constructs needed in business process applications.

Having fixed the specification language for the properties to check, in this paper we focus on the definition of a trace checking procedure for temporal properties. The definition of this procedure has to fulfill the following 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 — for checking the compliance of a system to its application requirements; R2) any procedure shall be scalable and enable checking of large traces within practical time limits, such that a trace with millions of events could be checked within seconds.

Requirement R1 emerges from the software development methodology embraced by our industrial partner, which has adopted MDE in practice and requires any software solution added to the development process (e.g., trace checking) to adhere to OMG specifications and rely on the corresponding tools. We remark that this requirement prevents the adoption of a naive approach for trace checking, in which *TemPsy* specifications would be first translated (likely manually, given the complexity of the task) into their corresponding temporal logic formulae and then verified using an existing trace checking approach (e.g., [4], [9]) optimized for temporal logic but not based on standard MDE technologies. Although this requirement is motivated by specific needs from our partner, we believe — based on experience — that it can be generalized to other contexts in which solutions have to be engineered by using standard MDE technologies that are already in place in the targeted development environment.

As for requirement R2, the trace checking procedure has to: 1) *scale with respect to the length of the trace*, because traces may contain a huge number of events, depending on the time span captured by the log, the nature of the system to which the log refers to (e.g., several virtual machines), and the types of events monitored (e.g., high-level message passing events or low-level method calls) [10]; 2) *complete within practical time limits*, because trace checking can be used not only for post-mortem analysis, but also to enable real-time log analysis (as a complementary strategy to run-time monitoring), to promptly detect critical requirements violations.

Keeping in mind the above requirements, the goal of this paper is to present *a scalable and practical solution, based on standard MDE technologies, for trace checking of pattern-based temporal properties expressed in* TemPsy. To achieve this goal, the paper will make the following contributions: i) a model-driven trace checking procedure, which relies on a mapping of temporal requirements written in *TemPsy* into Object Constraint Language (OCL) constraints on a meta-model of execution traces; ii) a publicly available tool (TEMPSY-CHECK) implementing this model-driven trace checking procedure; iii) an evaluation of the scalability of TEMPSY-CHECK, applied to the verification of real properties derived from a case study of our public service partner, including a comparison with a state-of-the-art alternative technology. As a separate contribution, we also make available the artifacts used in the evaluation to contribute to the building of a public repository of case studies for evaluating trace checking/run-time verification procedures.

Our trace checking procedure fulfills requirement R1 above since it follows a model-driven approach based on OCL. OCL is the de-facto constraints specification language defined by OMG and an international standard [11], which is supported by mature constraint checking technology, such as the constraint checker included in Eclipse OCL [12]. The procedure relies on a generic meta-model of system execution traces and leverages an optimized mapping of *TemPsy* properties into OCL constraints defined over this trace model. This mapping is optimized based on the structure of the *TemPsy* property to check, in order to achieve better performance. More specifically, we show how the problem of checking a *TemPsy* property over an execution trace (i.e., the *TemPsy* trace checking problem) can be reduced to evaluating an OCL constraint (derived from the *TemPsy* property to check and semantically-equivalent to it) on an instance of the trace model; this check is executed using standard OCL checkers.

To show the fulfillment of requirement R2 above, we extensively evaluated the scalability of the proposed trace checking procedure, by assessing the relationship among the checking time, the structural properties of a trace (e.g., length, distribution of events), and the type of property to check. We evaluated the scalability of our TEMPSY-CHECK tool on real properties extracted from our case study, on traces with length ranging from 100K to 1M. We also compared the performance of TEMPSY-CHECK with MONPOLY [13], a state-of-the-art alternative technology, selected from the participants to the "offline monitoring" track of the international Competition on Software for Runtime Verification [14], [15]. The experimental results show that TEMPSY-CHECK can load and analyze very large traces (with one million events) in about two seconds and that it scales linearly with respect to the length of the trace to check. The results also show that TEMPSY-CHECK in practice performs similarly to or better than the state-of-the-art, depending on the type of properties, confirming the feasibility and benefits of a model-driven approach for trace checking of temporal properties.

The rest of the paper is structured as follows. Section II provides some background concepts of *TemPsy*. Section III describes our model-driven approach for trace checking of *TemPsy* properties. Section IV reports on the evaluation conducted with TEMPSY-CHECK. Section V discusses related work. Section VI concludes the paper, providing directions for future work.

II. BACKGROUND: THE *TemPsy* LANGUAGE

Property specification patterns (PSPs) have been initially proposed by Dwyer et al. [7] in the late '90s in the context of formal verification, as a means to express recurring properties in a generalized form, which could be formalized in different specification languages. PSPs consists of eight parametrizable *patterns* ("absence", "universality", "existence", "bounded existence", "precedence", "response", "precedence chain", "response chain"), representing high-level abstractions of formal specifications, and five *scopes* ("globally", "before", "after", "between-and", "after-until"), which indicate the portions of a system execution in which a certain pattern should hold.

In previous work we proposed *OCLR* [6], a pattern-based, domain-specific language for the specification of temporal properties; the most recent version of the language is now called *TemPsy* (Temporal Properties made easy) [8]. The design of (*OCLR* and) *TemPsy* was driven by the analysis of the requirements of various applications implementing business process models in the context of eGovernment systems. The analysis showed that all the requirements could be expressed through Dwyer's PSPs, with some additional constructs. Hence, *TemPsy* was designed with the goal of supporting Dwyer's PSPs with the following extensions: 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 as well as their chain versions; 4) additional variants for the bounded existence and absence patterns.

The main concepts within a property expressed in *TemPsy* are those of *scope* and *pattern*. Scopes and patterns refer to events, which correspond to the actual events 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 the system time unit (as suggested in [16]). For space reasons, we only explain *TemPsy* informally, focusing on the supported patterns; readers are referred to the online report [8] for the complete definition of the (formal) syntax and semantics.

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

Universality. An event should occur across the entire execution trace.

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) in which the occurrence of the second event (respectively, block) depends on the occurrence of the first event (respectively, block). Based on this definition, we add 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, and the time distance between *A* and *B* should be at most 5 time units (specified right after '`preceding`'), and the time distance between *B* and *C* (denoted with a # symbol) should be at least 2 time units.

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, first block) leads to the occurrence of the second event (respectively, second block). Similarly to *precedence*, we add support for timing information to enable expressing the time distance between two adjacent events.

To exemplify, 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*".

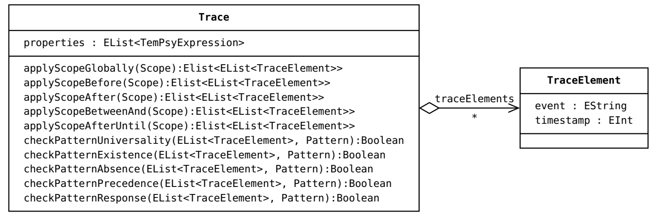

Fig. 1. Meta-model for execution traces

III. Model-driven Trace Checking of *TemPsy* properties

The idea at the basis of our model-driven trace checking approach is to *reduce* the problem of checking a *TemPsy* property ρ over a trace λ, to the problem of evaluating an OCL constraint (semantically equivalent to ρ) on an instance of a conceptual model for execution traces (equivalent to λ).

This reduction allows us to rely on standard constraint checking technology to perform trace checking; standard OCL checkers, such as Eclipse OCL, can be used to evaluate OCL constraints on model instances. The use of a model-driven approach and of standard technologies fulfills requirement R1 stated in section I, and enables us to provide a scalable and practical solution for trace checking of temporal properties, which is also viable in the long term.

A. *Meta-model for execution traces*

The definition of a meta-model for execution traces is a key element of our approach, since the transformation of *TemPsy* properties into efficiently checkable *OCL constraints defined on such model* is a key strategy for us to achieve scalability.

We propose a simple and yet generic meta-model of system execution traces; it can be extended (by enriching the type of event) depending on the actual type of system (e.g., business process, access control framework) and the type of properties to check. The model, depicted in Fig. 1 with a UML class diagram, contains a `Trace`, which is composed of a sequence of `TraceElements`, accessed through the association `traceElements`. Each `TraceElement` contains an attribute `event` of type string, which represents the actual event recorded in the trace, and an attribute `timestamp` of type integer, which indicates the time at which the event occurred. Class `Trace` contains also an attribute `properties`, which is a collection of `TemPsyExpressions`, representing the properties to be checked on the trace.

We have defined some side-effect-free operations in OCL for the `Trace` class; these operations consist of two types of functions. The first type, of the form `applyScope*S*`, are named after the different types of scope (e.g., `applyScopeBefore`, `applyScopeBetweenAnd`) and return segment(s) of a trace as determined by the parameters of the scope provided in input. The second type, of the form `checkPattern*P*`, are named after the different types of pattern (e.g., `checkPatternExistence`, `checkPatternPrecedence`) and check whether the pattern provided in input as the second parameter holds on the segment(s) represented by the first parameter.

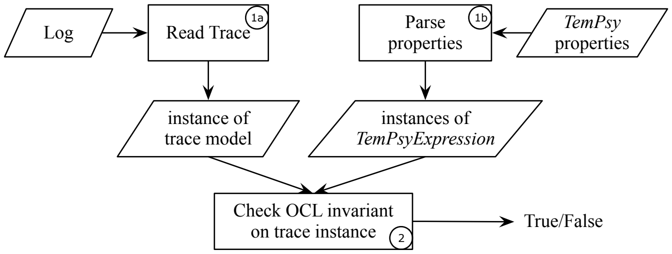

Fig. 2. Overview of the approach

B. Overview of the approach

Our approach for model-driven trace checking is sketched in Fig. 2: parallelogram shapes correspond to input/output artifacts, while rectangles correspond to steps in the approach. The two inputs are represented by a log, corresponding to the trace one wants to check, and by a set of *TemPsy* properties. The log file is read and converted (step 1a) to an instance of the class `Trace` in the model shown in Fig. 1. The *TemPsy* properties are parsed and converted (step 1b) to instances of class `TemPsyExpression`.

The key step (#2 in the figure) of our approach is to evaluate an OCL invariant on the trace instance. The checking of this invariant, which can be done using standard OCL checking tools, is semantically equivalent to performing trace checking of the *TemPsy* properties provided in input.

We have defined this invariant on the `Trace` class, as shown in Fig. 3. For every *TemPsy* property provided in input (and referenced in the instance of the trace through the attribute `self.properties`, line 2), the invariant evaluates a boolean function, which conceptually corresponds to applying the semantics of the pattern used in the property (accessed through the expression `property.pattern`) on a set of sub-traces, as defined by the scope used in the property (accessed through the expression `property.scope`).

More specifically, the body of the invariant expression is a multi-way branch (defined through a sequence of `if` statements), which selects a certain branch based on the specific scope type used within the property. Within the body of a branch, first a function of the form `applyScope*S*` is called. This function takes the scope used in the property as input and returns a collection of sub-traces, as defined by the scope semantics. Afterwards, the invariant invokes a function of the form `checkPattern*P*`, which checks whether the pattern used in the property holds on each sub-trace.

For instance, let us assume that the type of the scope of the *TemPsy* property provided in input is *globally* and that the type of the pattern used in the property is *response*. As shown in line 5, the function `applyScopeGlobally` is invoked to compute the sub-trace(s) defined by the `scope` parameter; the return value of this function is assigned to variable `subtraces`. The branch indicated on line 15 is then taken, which results in the evaluation of the boolean function `checkPatternResponse` on all the elements of `subtraces`, to check whether the input parameter `pattern` holds on each sub-trace.

```
1  context Trace
2  inv: self.properties->forAll(property:TemPsy::
       TemPsyExpression |
3   let scope:TemPsy::Scope = property.scope, pattern:
       TemPsy::Pattern = property.pattern in
4   if scope.type = TemPsy::GLOBALLY then
5     let subtraces:Sequence(OrderedSet(TraceElement)) =
         applyScopeGlobally(scope) in
6     if pattern.type = TemPsy::UNIVERSALITY then
7       subtraces->forAll(subtrace |
           checkPatternUniversality(subtrace, pattern))
8     else if pattern.type = TemPsy::EXISTENCE then
9       subtraces->forAll(subtrace |
           checkPatternExistence(subtrace, pattern))
10    else if pattern.type = TemPsy::ABSENCE then
11      subtraces->forAll(subtrace | checkPatternAbsence(
           subtrace, pattern))
12    else if pattern.type = TemPsy::PRECEDENCE then
13      subtraces->forAll(subtrace |
           checkPatternPrecedence(subtrace, pattern))
14    else if pattern.type = TemPsy::RESPONSE then
15      subtraces->forAll(subtrace | checkPatternResponse
           (subtrace, pattern))
16    endif endif endif endif endif
17  else if scope.type = TemPsy::BEFORE then [...]
18  else if scope.type = TemPsy::AFTER then [...]
19  else if scope.type = TemPsy::BETWEENAND then [...]
20  else if scope.type = TemPsy:AFTERUNTIL then [...]
21  endif endif endif endif endif)
```

Fig. 3. OCL invariant for checking *TemPsy* properties on a trace

The complete definitions in OCL of the functions of the form `applyScope*S*` and `checkPattern*P*` are available in the technical report accompanying this paper [8]. We illustrate examples of the `applyScope*S*` and `checkPattern*P*` functions in subsections III-C and III-D, respectively; to ease readability and conciseness, all the code snippets presented in these subsections are written using pseudocode.

C. Example of OCL functions for scopes (before)

To exemplify the OCL functions that are used to apply a scope definition on a trace, we illustrate the function `applyScopeBefore`, corresponding to the *before* scope. This function takes as input an object representing a scope in *TemPsy* and yields one or more segments of the trace, as determined by the semantics of the scope.

The pseudocode of the function `applyScopeBefore` is shown in Algorithm 1. The input parameter *scope* is an instance of the *before* scope, and the output is a list that contains the trace segments as determined by the structure of *scope*. We assume the parameter *scope* to have the form "`before [m] X [op n tu]`" (see section II), in which `op` stands for the comparison operator (i.e., "`at least`", "`at most`", or "`exactly`") used in the constraint that defines the time distance from the scope boundary event *X*.

The function starts by reading the parameters *X*, *m*, *op*, and *n* from the instance of the *before* scope (lines 1–4). In addition, we define and initialize to an empty list both variable *result* (to store the output value) and the auxiliary variable *segment*

Algorithm 1: applyScopeBefore

```
   Input: scope : an instance of the before scope structured as
          "before [m] X [op n tu]"
   Output: result : a list containing the trace segment as
           determined by the parameters of scope
1  X ← event name of the scope boundary
2  m ← index of the specific occurrence of event X
3  op ← comparison operator of the constraint on time distance
4  n ← time distance from the m-th occurrence of X
5  result ← [], segment ← []
6  if m = null then m ← 1
7  t ← timestamp of the m-th occurrence of event X
8  if t ≠ null then
9      switch op do
10         case "at least" do
11             segment ← trace elements with timestamp t′
                 satisfying t′ ≤ t − n
12         case "at most" do
13             segment ← trace elements with timestamp t′
                 satisfying t − n ≤ t′ < t
14         case "exactly" do
15             segment ← trace elements with timestamp equal to
                 t − n
16         otherwise do
17             segment ← trace elements with timestamp t′
                 satisfying t′ < t
18 result.append(segment)
19 return result
```

Algorithm 2: checkPatternPrecedenceAtLeast

```
Input: a trace segment subtrace and the parameters of an
       instance of precedence pattern of the form "block1
       preceding at least n tu block2": two events (chains)
       block1 and block2, and a threshold n of the time
       distance between block1 and block2
Output: true if pattern holds on subtrace; false otherwise
size1, size2 ← the sizes of block1 and block2
firstOfBlock1 ← block1.first().event
firstOfBlock2 ← block2.first().event
(i1, pt1) ← (1, 0), (i2, pt2) ← (1, 0)
flag1 ← true
for elem ∈ subtrace do
    e ← elem.event
    t ← elem.timestamp
    if flag1 then
        if e = firstOfBlock1 then (i1, pt1) ← (2, t)
        else if i1 > 1 then (i1, pt1) ← match(block1, i1, pt1, e, t)
        if i1 = size1 + 1 then flag1 ← false
    if e = firstOfBlock2 then
        if flag1 ∥ t < pt1 + n then (i2, pt2) ← (2, t)
        else return true
    else if i2 > 1 then (i2, pt2) ← match(block2, i2, pt2, e, t)
    if i2 = size2 + 1 then return false
return true
```

(for collecting intermediate trace elements). If the parameter m is omitted in the scope definition, variable m is replaced with the value 1 (line 6), according to the default semantics of the *before* scope. We then assign to variable t the timestamp of the m-th occurrence of event X in the trace (line 7). If t is defined, it means that the m-th occurrence of the event has been found in the trace. Lines 9–17 select a segment from the trace, based on the value of *op*. For example, when *op* is "`at least`", line 11 selects all the trace elements that occur at least n time unit(s) before the m-th occurrence of event X. If no time distance constraint is specified in the *scope* (line 17), the function selects the trace segment starting at the beginning of the trace and ending at the m-th occurrence of event X. The function ends by adding the *segment* selected from the trace to the output variable *result*.

D. *Example of OCL functions for patterns (*precedence*)*

To exemplify the OCL functions that are used to check if a pattern holds on a sub-trace, we present the function `checkPatternPrecedence`, corresponding to the *precedence* pattern. This function takes as input a sub-trace and an object representing a pattern in *TemPsy*, and returns whether the pattern holds on the input sub-trace. The definition of function `checkPatternPrecedence` comes in four variants, to consider the case where no time distance is specified between the two blocks of the patterns, and the three cases with the different comparison operators (i.e., "`at least`", "`at most`", and "`exactly`"). For space reasons, we only describe the function `checkPatternPrecedenceAtLeast`, shown in Algorithm 2.

This function takes as input a trace segment and the parameters of an instance of a *precedence* pattern: $block_1$, $block_2$, and the optional time distance n between them. Notice that $block_1$ and $block_2$ can be either an atomic event or a chain of events with optional constraints on the time distances in between. The semantics of the pattern prescribes that each occurrence of $block_2$ is preceded, possibly with a certain time distance, by an occurrence of $block_1$. In practice, we need to check whether there is an occurrence of $block_1$ before the *first* occurrence of $block_2$ (and at a certain time distance, if required), since this implies that any other occurrence of $block_2$ occurring after the first one is preceded by an occurrence of $block_1$. We report a violation if we cannot find an occurrence of $block_1$ before the first occurrence of $block_2$ or if the distance between the two blocks is less than n.

The algorithm uses several auxiliary variables: $size_1$ and $size_2$ keep track of the number of events to match in each block; *firstOfBlock1* and *firstOfBlock2* contain the event of each block's first element. The integer tuple (i_1, pt_1) (respectively (i_2, pt_2)) is used to determine whether the trace element being checked is a match of the next event in $block_1$ (respectively, $block_2$). More specifically, element i_1 (respectively, i_2) stores the position within $block_1$ (respectively, $block_2$) of the next event to be matched; element pt_1 (respectively, pt_2) stores the timestamp of the previous trace element matched at $block_1[i_1 - 1]$ (respectively, $block_2[i_2 - 1]$). The boolean $flag_1$ is initialized to `true` and is set to `false` when the first occurrence of $block_1$ has been fully matched.

Algorithm 3: match

Input: an events chain *block*, a tuple (i, pt) of which i ($i > 1$) stores the position (within *block*) of the event to be checked, pt stores the timestamp of the previous trace element if it was a match for $block[i-1]$, and a trace element (e, t) to be matched with $block[i]$

Output: $(i+1,t)$ if the trace element is a match for $block[i]$; $(1,0)$ otherwise

```
1 if e = block[i].event then
2     op ← block[i].timeDistance.op
3     t' ← pt + block[i].timeDistance.value
4     if compare(t, op, t') then (i,pt) ← (i+1,t)
5 else (i,pt) ← (1,0)
6 return (i, pt)
```

The function contains a loop that iterates through all the elements of the input *subtrace*, trying to match each element with $block_1[i_1]$ (lines 9–12) and with $block_2[i_2]$ (lines 13–17).

As for matching $block_1$, if the current trace element matches the first event of $block_1$ (line 10), the variable i_1 is set to 2 and pt_1 is updated with the current timestamp. Otherwise, if the next event of $block_1$ to be matched is not the first, an auxiliary function `match` (shown in Algorithm 3) is called to match the event defined at i_1.

Function `match` takes as input five parameters: an events chain *block*, two integer parameters i and pt, of which i ($i > 1$) stores the position (within *block*) of the event to be checked and pt stores the timestamp of the previous trace element (if it was a match for $block[i-1]$), and the two parameters of a trace element (e,t) to be matched with $block[i]$. The function updates the tuple (i,pt) if the input element is a match for $block[i]$; else it sets the tuple to $(1,0)$. More specifically, if the current element is an occurrence of the event defined at $block[i]$ (with $i > 1$) (line 1), and if the constraint on the distance (if defined[1]) from the previous event at $block[i-1]$ holds (line 4), variable i is incremented and variable pt is updated with the timestamp of current trace element (line 4). Otherwise, the tuple (i,pt) is reset on line 5.

At line 12 of function `checkPatternPrecedenceAtLeast`, if the matched event is the last event of $block_1$ (meaning that an occurrence of *block1* has been found preceding any possible occurrence of *block2*), variable $flag_1$ is set to `false`.

As for matching $block_2$, if the occurrence of the *first* event of $block_2$ is detected (line 13), there are two cases that may lead to a violation. Either $block_1$ has not been fully matched yet (i.e., variable $flag_1$ is `true`) or it has been fully matched but the timestamp of the current trace element (matching the first element of $block_2$) violates the constraint on the distance between $block_1$ and $block_2$. If one of these two conditions holds (line 14), the algorithm goes on to match[2] the rest of $block_2$ (line 16), since the current element might actually not be part of a whole instance of $block_2$. If both of these conditions are not satisfied (line 15), it means that there is no violation, i.e., the first block has been fully matched and the distance constraint between the two blocks is satisfied; hence, there is no need to match[3] the remainder of $block_2$ and the algorithm returns `true`. Otherwise the algorithm invokes function *match* to match the current element with $block2[i_2]$ (line 16). The function reports a violation whenever $block_2$ is fully matched (line 17); otherwise, it returns `true` after the loop (line 18).

E. Tool Implementation

We have implemented our model-driven approach for trace checking of *TemPsy* properties in a tool named TEMPSY-CHECK. The tool is based on Xtext [17] and Eclipse OCL; it is publicly available online [18].

TEMPSY-CHECK takes as input a log file in CSV format and converts it to an intermediary representation (called "trace description"), defined as a domain-specific language using the Xtext framework. We have introduced this intermediate representations for traces to support, in the future, multiple input raw formats for trace logs. The trace description is then used to generate an XMI file corresponding to an instance of the trace model. The tool also takes as input a list of *TemPsy* properties and converts them into an XMI-based format. The evaluation of the OCL constraints corresponding (as described in the previous subsections) to the properties to check on the trace is done using the OCL checker included in Eclipse OCL, whose output (true/false) is then returned to the user.

IV. EVALUATION

A. Overview, Methodology, and Settings

The evaluation of TEMPSY-CHECK focuses on its scalability, since trace checking tools are expected to be able to handle very large traces. Indeed, traces may contain a huge number of events, depending on the time span captured by the log, the nature of the system to which the log refers to (e.g., several virtual machines), the types of events monitored (e.g., high-level message passing events or low-level method calls) [10]. In the evaluation, we assess the relationship between the time taken to check a property on a trace and the structural properties of the trace (e.g., length, distribution of events) and the type of property to check; we also compare the performance of TEMPSY-CHECK with a state-of-the-art alternative technology.

We have conducted our evaluation using a benchmark consisting of a subset of the properties extracted from the requirements specification documents of one of the eGovernment applications developed by our public service partner. Out of the 47 properties documented in the case study, in this paper, for space reasons, we report only on the evaluation with the 12 properties using a *globally* scope[4]. They are the

[1]The pseudocode for dealing with the case when the distance between block elements is not defined has been omitted for simplicity.

[2]Notice that in this case a violation is reported only if $block_2$ is fully matched (line 17).

[3]This is derived from the formal semantics of the `preceding` operator, in which the match of the first block, at the proper time distance, is defined as the consequent of the logical implication that formalizes the semantics of the operator.

[4]The complete evaluation report and the data related to the remaining properties (defined using other scopes) are available in [8].

TABLE I
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*

most challenging in terms of scalability, since the semantics of this scope guarantees that the pattern (used in the property to check) will be evaluated through the entire length of the trace. The 12 properties used for the evaluation are listed in a sanitized form in Table I. The actual textual description of each property has been omitted for confidentiality reasons; the events involved in the property (e.g., "a citizen requests a certificate") are denoted using uppercase letters.

These properties have been checked on *synthesized* traces. We use synthesized traces instead of real ones because: 1) based on our experience, real traces are often inadequate to cover a large range of trace lengths and a variety of properties; 2) we wanted to have great diversity in terms of occurrences of patterns in the traces, while being able to control this diversity; 3) real traces are valuable to assess fault detection capabilities, while in our case we focus on the scalability of the approach; 4) if we had used real traces, they could not be shared for forming a public benchmark, even when sanitized. By using synthesized traces we are able to control in a systematic way the factors (such as trace length, sub-trace(s) length and position, frequency and distance of events) that could impact the execution time for a specific type of property. At the same time, we are also able to randomly set other factors, to avoid any bias.

To synthesize these traces we implemented a trace generator program. This program allows for generating diverse (in terms of size, patterns, scopes, event positions and frequency) and realistic traces, without introducing bias. The generator takes as input a property, the desired length of the trace to generate and additional parameters depending on the type of property given in input and the factors one wants to control. To determine the position in the trace of the events occurring in the input property, the generator takes into account the temporal and timing constraints prescribed by the semantics of the scope and the pattern used in the property. Positions in the trace that are deemed not relevant for the evaluation of the property are filled with a dummy event. The trace generation strategy depends on the scope and pattern used in each property and is discussed in detail below. As an additional contribution of the paper, we also make available in the TEMPSY-CHECK GitHub repository [18] the artifacts used in the evaluation, to contribute to the building of a public repository of case studies for evaluating trace checking/run-time verification procedures.

Moreover, to assess scalability, we also need a baseline of comparison. Such baseline should be the best available tool that can be considered an alternative to TEMPSY-CHECK. We identified such a tool among the participants to the "offline monitoring" track of the 2014 and 2015 international Competition on Software for Runtime Verification (CSRV 2014 [14] and CSRV 2015 [15]). Out of the tools (LOGFIRE [19], MARQ/QEA [20], MONPOLY [13], RITHM2 [21], RV-MONITOR [22], STEPR) qualified for the final round of the two editions of the competition, LOGFIRE, RITHM2, and STEPR were not publicly available[5] at the time of writing. Among the remaining three, MARQ/QEA does not support any input language and uses an automata-based formalism: the user has to write a Java program that builds the automaton corresponding to the property to check; on the other hand, both MONPOLY and RV-MONITOR support a specification language that is conceptually close to *TemPsy*. We chose MONPOLY over RV-MONITOR because it achieved a better score (293.54 vs 265.39). MONPOLY supports MFOTL, a metric first-order temporal logic, as specification language; to perform the comparison with it, we manually translated the properties into MFOTL formulae. These formulae are also available in the TEMPSY-CHECK GitHub repository [18]. We remark that our goal, in this comparison, is not necessarily to fare better than existing technology, but to verify that an MDE approach to trace checking is viable from a scalability standpoint.

The results reported in this section have been measured on a desktop computer with a 3 GHz Intel Dual-Core i7 CPU and 16GB 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), Eclipse OCL v. 6.0.1, and MONPOLY v. 1.1.6. All measurements reported correspond to the average value over 100 runs of the check procedure (on the same trace, for the same property).

B. Scalability analysis

To assess the scalability of our approach, we address the following research questions:

RQ-G1) What is the relation between the execution time of the trace checking procedure and the length of a trace?

RQ-G2) What are the types of pattern most taxing on the execution time?

RQ-G3) How does TEMPSY-CHECK compare with MONPOLY in terms of execution time?

1) Trace Generation Strategy: In the case of the *globally* scope the generation of the trace is determined only by the semantics of the pattern used in the property.

For the *universality* pattern, we repeat the event occurring in it through the entire trace.

For the *existence* pattern, we first determine the number n of occurrences to generate, based on the bound indicated in the

[5]The first version of RITHM is available but it only supports run-time verification of C programs. As for STEPR, no reference is available in the competition report [14] or online.

property. If the bound is expressed as "`at least` m" or "`at most` m" we randomly generate n with a uniform distribution on the range $[m, \text{trace length}]$, respectively $[0, m]$; if the bound is expressed as "`exactly` m", n is set to m. Afterwards, we randomly generate (with a uniform distribution on the range $[1, \text{trace length}]$) n positions in the trace where to put the occurrences of the event specified in the property.

For the *absence* pattern, if the property has the form `never A`, the trace is generated without any occurrence of the event A. If the property has the form `never exactly` m `A`, we randomly generate n with a uniform distribution on the range $[0, \dots, m-1, m+1, \dots, \text{trace length}]$.

In the case of a property containing a *precedence* or *response* pattern, we generate a number of occurrences of events (involved in the property) equal to 10% of the length of the trace. This value has been selected based on the frequency of events observed in the application whose requirements are expressed through the properties shown in Table I. The simplest case is for a property like `globally B responding at most 10 tu A`: assuming a trace length of 1M, we would generate 50K occurrences of the pattern (i.e., pairs of `A` and `B`), for a total of 100K occurrences of `A` and `B`. More complex cases have to take into account the event chains used in the property. For the distribution of the occurrences of the pattern across the trace we have to consider the distance between events. For example, for the property aforementioned, each occurrence of the response pattern would span over at most 10 time units; this is the maximum distance between an occurrence of `A` and the corresponding occurrence of `B`. The number of pattern occurrences to generate and the maximum time span of each pattern occurrence are the parameters used to randomly allot the pattern occurrences over the trace, according to a uniform distribution.

2) Evaluation: We run the trace checking procedure for properties P1–P12; each property was checked on ten different traces, with length (i.e., number of events) varying from 100K to 1M. The twelve plots in Fig. 4 depict the execution time of TEMPSY-CHECK (denoted by ○) and of MONPOLY (denoted by $*$) for each of the properties P1–P12, for different trace lengths. The execution time for both tools has been measured using the `time` Unix command.

We answer RQ-G1 by observing that the time taken by TEMPSY-CHECK ranges from about one hundred milliseconds to a bit more than two seconds, and increases linearly with the length of the trace, depending on the type of property. To answer RQ-G2, the results show that the properties more taxing on the execution time are those with a *response* or *precedence* pattern (e.g., P5, P6, P7, P9, P11). Regarding RQ-G3, we observe that the time taken by MONPOLY ranges from about one hundred milliseconds to a bit less than eight seconds, and is also linear with respect to the length of the trace. MONPOLY takes longer for checking properties with a *(bounded) existence* pattern (e.g., P3, P4) and with a *precedence* pattern that contains a distance constraint of type "`at least`" (e.g., P10). We can answer RQ-G3 stating that, except for the case of properties P3, P4, and P10, the two tools perform almost similarly, with absolute differences between execution times that are quite small (less than one second). In the case of properties P3, P4, and P10, TEMPSY-CHECK performs much better than MONPOLY. A possible explanation for the slower time of MONPOLY for these properties could be the structure of the corresponding MFOTL formulae, which contain several nested temporal operators to express the "eventually at least/at most" pattern (P3, P4) and an event chain (P10).

The execution times discussed above include not only the time to perform the actual check, but also the time to parse/load the trace to check[6]. The average trace loading time for TEMPSY-CHECK, measured through instrumentation, ranges from 55 ms to 550 ms, growing linearly with respect to the trace length. Notice that for checking a single property on a trace with TEMPSY-CHECK, the trace loading time can take, for larger traces, from one-fourth to one-third of the total execution time. Although these values for the trace loading time can seem high, we expect the loading time not to impact on the total execution time in the case of *batch property checking*, i.e., checking multiple properties at the same time on a trace. Checking in batch mode a set of properties, rather than individual ones, is common in enterprise scenarios in which, for example, the set of properties to check is decided by the entity that has invoked a business process [23].

To further investigate this aspect, we compared the execution time of TEMPSY-CHECK and MONPOLY for batch checking ten properties (P3–P12), over ten traces, with length ranging from 1M to 10M. These traces have been obtained by concatenating the traces used for the experiment described above, and by renaming the events within each trace being concatenated, to avoid name clashes. We executed TEMPSY-CHECK by providing in input the list of the ten properties to check. We executed MONPOLY by providing in input one formula corresponding to the conjunction of the ten formulae equivalent to properties P3–P12. Figure 5 shows the result of the comparison: the performance of the two tools are similar for traces of length up to six millions; over this threshold, MONPOLY gets slower.

C. Discussion

The results presented above (as well as the additional data presented in the technical report accompanying this paper [8]) show the feasibility and benefits of applying our model-driven approach for trace checking in realistic settings.

Our TEMPSY-CHECK tool is a viable technology from a performance standpoint point as it can load and analyze very large traces (with one million events) in about two seconds. The tool scales linearly with respect to the length of the input trace to check. Notice that "the input trace to check" can correspond also to a sub-trace of an actual, larger execution trace. This can be the case for properties referring to events occurring in time windows (see, for example, the service provisioning patterns presented in [24]). In these cases, one

[6] The trace loading time is not available in the output of MONPOLY.

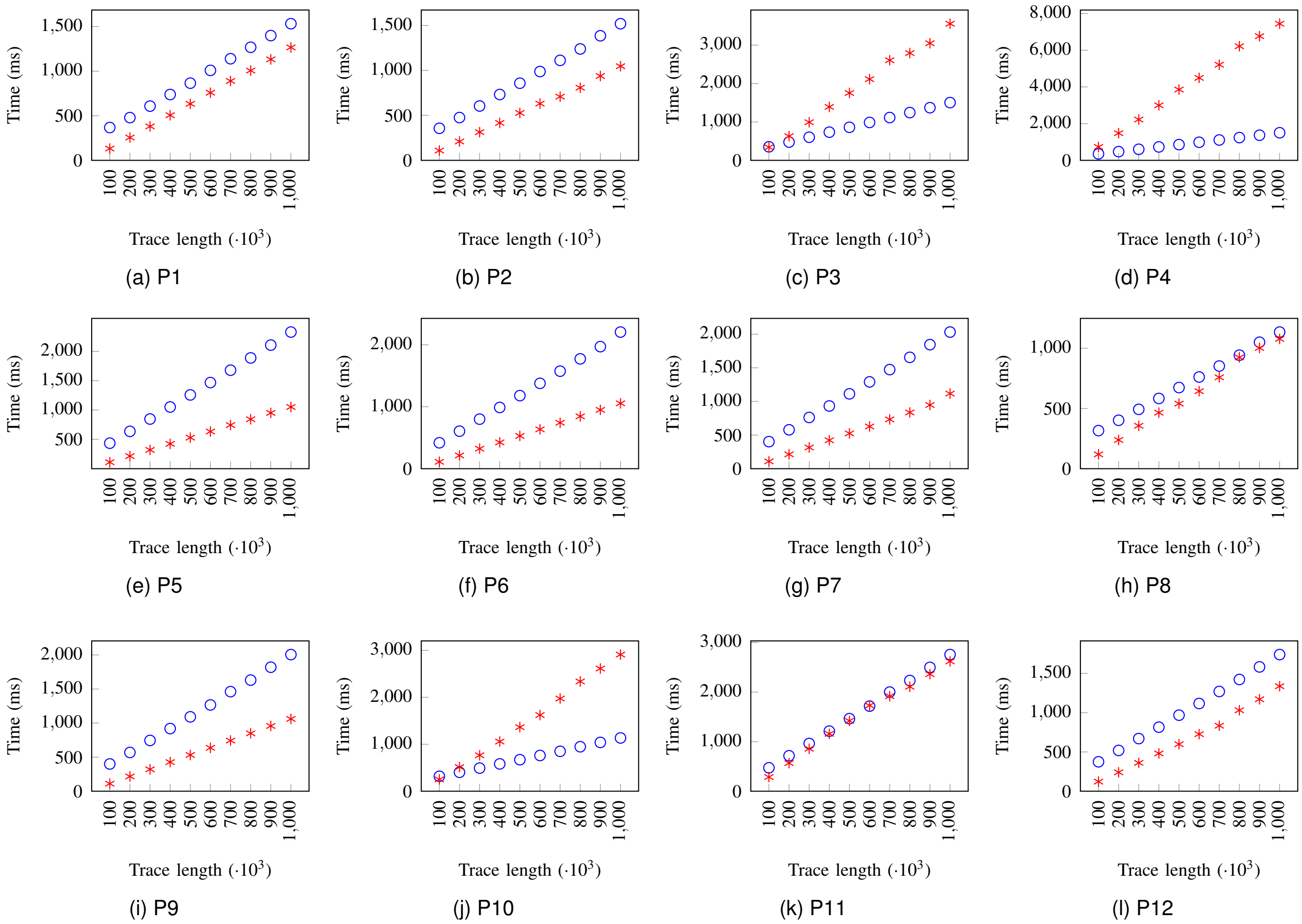

Fig. 4. Comparison between the execution time of TEMPSY-CHECK (○) and of MONPOLY (∗) for properties with the *globally* scope

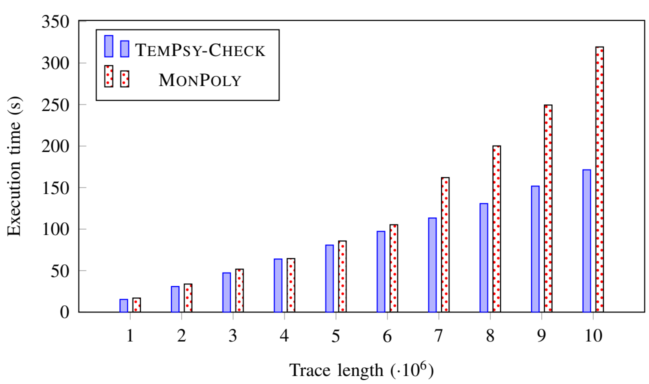

Fig. 5. Comparison of the execution time for the batch checking of ten properties with the *globally* scope

can first isolate from the original trace the window of interest and then feed the latter to our tool.

We have also compared the performance of our implementation to MONPOLY, a comparable, state-of-art tool. Despite the fact that MONPOLY is a tool that implements a dedicated algorithm [4] for trace checking of temporal logic properties, our TEMPSY-CHECK tool (which relies on a generalist OCL checker) not only achieves similar results, but in some cases it also performs better than MONPOLY.

We also remark that writing some of the properties in MFOTL was challenging (despite previous knowledge of MFOTL), much more than when using *TemPsy*. This challenge could be overcome by defining properties in *TemPsy* and then providing an automatic translation to MFOTL formulae or, dually, by building a system of property specification patterns on top of MFOTL. In both cases, one could have relied on MONPOLY for trace checking. While this could be in principle a viable approach, it would not fulfill requirement R1 (see section I), which entails to rely on standard constraint checking technology — complying with OMG specifications — for checking temporal properties. We remark that this requirement is not specific to this project, but is more general because there are many contexts where solutions have to be engineered by using standardized MDE technologies.

Overall, we can conclude that a model-driven approach to trace checking of realistic temporal properties is viable, even on very large traces, and performs similarly to or better than the state-of-the-art, depending on the type of properties.

Threats to validity: The main threat to validity to the results presented above is the intrinsic presence of errors in the toolchain we developed, which might not reflect the semantics of *TemPsy*. We tried to compensate for this by thoroughly testing the checker with traces and properties for which the oracle 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 events occurrences and time distances. However, this threat does not affect our research question on scalability, as we want to analyze the execution time as a function of a number of parameters (e.g., trace length), while varying randomly other aspects (e.g., position of certain events). As explained at the beginning of this section, for that purpose, 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 checker, with lower performance. We chose Eclipse OCL for its scalability (see [25]). Finally, as for the comparison with MONPOLY, we remark that its specification language (MFOTL) is more expressive than *TemPsy* (e.g., by supporting first-order quantification), hence the performance of MONPOLY could have been negatively affected by the more complex implementation needed to support a richer specification language. Moreover, the MFOTL properties that we wrote to perform the comparison described in subsection IV-B could be written in a different, but semantically-equivalent form that could lead to different results. We tried to mitigate this aspect by having the MFOTL formulae written by a person with ten years of experience in formal specification (and verification) with temporal logics. Furthermore, we believe that in practice, it might be hard anyway for practitioners (with limited background in temporal logic) to find out what is the optimal way to express a property in MFOTL.

V. RELATED WORK

Model-driven technologies have been used in various work on (run-time) trace and/or assertion checking. The model-driven approach for assertion checking proposed in [26] relies on the principles of aspect-oriented programming and uses a technique called two-level aspect weaving. First, cross-cutting assertions defined using ECL, an extension of OCL, are weaved into a model defined within GME (Generic Modeling Environment [27]) and then the code for checking the contracts specified in the models is generated using model-driven program transformations [28]. ECL does not support the expression of temporal constraints. An approach conceptually similar to ours is proposed in [29], in which pre- and post-conditions are expressed with visual contracts defined using graph transformations and then transformed into a code-level representation as JML (Java Modeling Language) assertions. The pre- and post-conditions that can be expressed in this framework are functional and do not support temporal expressions. The approach for model-driven monitoring of Web services proposed in [30] considers temporal properties expressed using property specification patterns [7] and defined with a subset of UML 2.0 Sequence Diagrams; these properties are checked at run time by translating sequence diagrams into non-deterministic finite automata. However, these properties, differently from those that can be expressed with *TemPsy*, do not support expressing timing requirements. Our model-driven approach for trace checking can be easily applied in scenarios where other trace models are used, as long as OCL invariants can be expressed on them; examples of these models are those proposed in [31] (for the reverse engineering of UML sequence diagrams from traces) and [32] (tailored for the exchange of traces corresponding to large program call trees).

This work is also related to the more general area of trace checking/run-time verification [33]. The majority of the approaches proposed in this area — for example, [4], [34]–[36], including previous work of some of the authors [9], [10], [37] — focuses on the verification of temporal properties expressed using some temporal logic. These approaches define the trace checking/run-time verification problem in terms of a *word problem*, i.e., the problem of whether a given word is included in some languages, and rely on formal verification tools like model checkers or SAT/SMT solvers. In our approach, we use a domain-specific specification language (*TemPsy*) and rely on standard constraint checking technology complying with OMG specifications.

VI. CONCLUSION AND FUTURE WORK

Trace checking is a procedure for checking the compliance of a system with respect to its requirements, by analyzing the log of events produced by the system during its execution. In this paper we have presented a scalable and practical solution for trace checking of the temporal requirements expressed using a pattern-based specification language. Our solution can be used in contexts where: model-driven engineering is already a practice; relying on standards and industry-strength tools for property checking is a fundamental prerequisite; the checking procedure should scale with respect to the length of the trace, to allow checking very large traces, and should complete within practical time limits, to enable real-time log analysis.

The results of the evaluation show the feasibility and benefits of applying our model-driven approach for trace checking in realistic settings. TEMPSY-CHECK can load and analyze very large traces (with one million events) in about two seconds; it scales linearly with respect to the length of the trace to check. The results also show that TEMPSY-CHECK in practice performs similarly to or better than the state-of-the-art, depending on the type of properties.

As part of future work, we plan to extend TEMPSY-CHECK to provide a more informative output than the boolean result currently returned when violations are detected in a trace, by adding support for interactive inspection of violations.

ACKNOWLEDGEMENT

This work has been supported by the National Research Fund, Luxembourg (FNR/P10/03). The authors would like to thank the members of the Prometa team at CTIE.

References

[1] A. Mrad, S. Ahmed, S. Hallé, and E. Beaudet, "BabelTrace: A collection of transducers for trace validation," in *Proc. RV 2012*, ser. LNCS, vol. 7687. Heidelberg, Germany: Springer, 2013, pp. 126–130.

[2] M. Felder and A. Morzenti, "Validating real-time systems by history-checking TRIO specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 3, no. 4, pp. 308–339, Oct. 1994.

[3] W. Dou, D. Bianculli, and L. Briand, "Revisiting model-driven engineering for run-time verification of business processes," in *Proc. SAM 2014*, ser. LNCS, vol. 8769. Heidelberg, Germany: Springer, September 2014, pp. 190–197.

[4] D. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann, "Runtime monitoring of metric first-order temporal properties," in *Proc. FSTTCS '08*, vol. 2. Dagstuhl, Germany: IBFI Schloss Dagstuhl, 2008, pp. 49–60.

[5] D. Bianculli, C. Ghezzi, and P. San Pietro, "The tale of SOLOIST: a specification language for service compositions interactions," in *Proc. FACS'12*, ser. LNCS, vol. 7684. Heidelberg, Germany: Springer, 2013, pp. 55–72.

[6] W. Dou, D. Bianculli, and L. Briand, "OCLR: a more expressive, pattern-based temporal extension of OCL," in *Proc. ECMFA 2014*, ser. LNCS, vol. 8569. Heidelberg, Germany: Springer, July 2014, pp. 51–66.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. ICSE 1999*. New York, NY, USA: ACM, 1999, pp. 411–420.

[8] W. Dou, D. Bianculli, and L. Briand, "A model-based approach to offline trace checking of temporal properties with OCL," SnT Centre - University of Luxembourg, Tech. Rep. TR-SnT-2014-5, September 2014. [Online]. Available: http://hdl.handle.net/10993/16112

[9] M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krstić, and P. San Pietro, "SMT-based checking of SOLOIST over sparse traces," in *Proc. FASE 2014*, ser. LNCS, vol. 8411. Heidelberg, Germany: Springer, April 2014, pp. 276–290.

[10] D. Bianculli, C. Ghezzi, and S. Krstić, "Trace checking of metric temporal logic with aggregating modalities using MapReduce," in *Proc. SEFM 2014*, ser. LNCS, vol. 8702. Heidelberg, Germany: Springer, September 2014, pp. 144–158.

[11] OMG, "ISO/IEC 19507 (OCL v2.3.1)," http://www.omg.org/spec/OCL/ISO/19507/PDF, April 2012.

[12] Eclipse, "Eclipse OCL tools," http://www.eclipse.org/modeling/mdt/?project=ocl, Sep. 2015.

[13] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu, "MONPOLY: Monitoring usage-control policies," in *Proc. RV 2011*, ser. LNCS, vol. 7186. Heidelberg, Germany: Springer, 2012, pp. 360–364.

[14] E. Bartocci, B. Bonakdarpour, and Y. Falcone, "First international competition on software for runtime verification," in *Proc. RV 2014*, ser. LNCS. Heidelberg, Germany: Springer, 2014, vol. 8734, pp. 1–9.

[15] Y. Falcone, D. Ničković, G. Reger, and D. Thoma, "Second international competition on runtime verification," in *Proc. RV 2015*. Heidelberg, Germany: Springer, 2015, pp. 405–422.

[16] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *Proc. ICSE '05*. ACM, 2005, pp. 372–381.

[17] Eclipse, "Xtext–Language Engineering Made Easy!" http://www.eclipse.org/Xtext/, Nov. 2015.

[18] W. Dou. Tempsy-check tool web site. http://weidou.github.io/TemPsy-Check/.

[19] K. Havelund, "Rule-based runtime verification revisited," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 2, pp. 143–170, Apr. 2015.

[20] G. Reger, H. C. Cruz, and D. Rydeheard, "Marq: Monitoring at runtime with qea," in *Profc. TACAS 2015*. Heidelberg, Germany: Springer, 2015, pp. 596–610.

[21] S. Navabpour, Y. Joshi, W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister, "RiTHM: A tool for enabling time-triggered runtime verification for C programs," in *Proc. ESEC/FSE 2013*. New York, NY, USA: ACM, 2013, pp. 603–606.

[22] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, and G. Rosu, "RV-Monitor: Efficient parametric runtime verification with simultaneous properties," in *Proc. RV'14*, ser. LNCS. Heidelberg, Germany: Springer, September 2014, pp. 285–300.

[23] L. Baresi and S. Guinea, "Towards dynamic monitoring of WS-BPEL processes," in *Proc. ICSOC 2005*, ser. LNCS, vol. 3826. Heidelberg, Germany: Springer, 2005, pp. 269–282.

[24] D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti, "Specification patterns from research to industry: a case study in service-based applications," in *Proc. ICSE 2012*. Piscataway, NJ, USA: IEEE, 2012, pp. 968–976.

[25] I. Raáth and E. Willink, "Fast, faster and super-fast queries," http://www.eclipse.org/modeling/mdt/ocl/docs/publications/EclipseConEurope2012/FastQueries.pdf, eclipseCon Europe 2012 presentation.

[26] J. Zhang, J. Gray, and Y. Lin, "A model-driven approach to enforce crosscutting assertion checking," in *Proc. MACS '05*. New York, NY, USA: ACM, 2005, pp. 1–5.

[27] J. Davis, "GME: The generic modeling environment," in *Companion of the Proc. of OOPSLA '03*. New York, NY, USA: ACM, 2003, pp. 82–83.

[28] J. Gray, J. Zhang, Y. Lin, S. Roychoudhury, H. Wu, R. Sudarsan, A. Gokhale, S. Neema, F. Shi, and T. Bapty, "Model-driven program transformation of a large avionics framework," in *Proc. GPCE 2004*, ser. LNCS, vol. 3286. Heidelberg, Germany: Springer, 2004, pp. 361–378.

[29] G. Engels, M. Lohmann, S. Sauer, and R. Heckel, "Model-driven monitoring: An application of graph transformation for design by contract," in *Proc. ICGT 2006*, ser. LNCS, vol. 4178. Heidelberg, Germany: Springer, 2006, pp. 336–350.

[30] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse, "Runtime monitoring of web service conversations," *IEEE Trans. Serv. Comput.*, vol. 2, no. 3, pp. 223–244, 2009.

[31] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of UML sequence diagrams for distributed Java software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 642–663, Sep. 2006.

[32] A. Hamou-Lhadj and T. C. Lethbridge, "A metamodel for the compact but lossless exchange of execution traces," *Softw. Syst. Model.*, vol. 11, no. 1, pp. 77–98, Feb. 2012.

[33] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, May/June 2009.

[34] B. Finkbeiner, S. Sankaranarayanan, and H. Sipma, "Collecting statistics over runtime executions," *Form. Method Syst. Des.*, vol. 27, pp. 253–274, 2005.

[35] D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu, "Monitoring of temporal first-order properties with aggregations," in *Proc. RV 2013*, ser. LNCS, vol. 8174. Heidelberg, Germany: Springer, 2013, pp. 40–58.

[36] B. Barre, M. Klein, M. Soucy-Boivin, P.-A. Ollivier, and S. Hallé, "MapReduce for parallel trace validation of LTL properties," in *Proc. RV 2012*, ser. LNCS, vol. 7687. Heidelberg, Germany: Springer, 2013, pp. 184–198.

[37] M. Bersani, D. Bianculli, C. Ghezzi, S. Krstić, and P. San Pietro, "Efficient large-scale trace checking using MapReduce," in *Proc. ICSE 2016*. New York, NY, USA: ACM, May 2016, pp. 888–898.