Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

A Model-driven Approach to Trace Checking of Temporal Properties with Aggregations

Chaima Boufaied[a] Domenico Bianculli[a] Lionel Briand[a]

a. SnT Centre - University of Luxembourg, Luxembourg, Luxembourg

Abstract The verification of complex software systems often requires to check quantitative properties that rely on aggregation operators (e.g., the average response time of a service). One way to ease the specification of these properties is to use property specification patterns, such as the ones for "service provisioning", previously proposed in the literature.

In this paper we focus on the problem of performing offline trace checking of temporal properties containing aggregation operators. We first present *TemPsy-AG*, an extension of *TemPsy*—an existing pattern-based language for the specification of temporal properties—to support service provisioning patterns that use aggregation operators. We then extend an existing model-driven procedure for trace checking, to verify properties expressed in *TemPsy-AG*. The trace checking procedure relies on the efficient mapping of temporal properties written in *TemPsy-AG* into OCL constraints on a meta-model of execution traces. We have implemented this procedure in the TemPsy-Check-AG tool and evaluated its performance: our approach scales linearly with respect to the length of the input trace and can deal with much larger traces than a state-of-the-art tool.

Keywords Temporal properties; Offline trace checking; Aggregation operators; Specification patterns; OCL.

1 Introduction

Trace checking is a *run-time verification (RV)* [LS09] technique that checks whether a property holds over a log (i.e., a trace) of recorded events. Since the log of events is obtained and checked *after* the execution of a system terminates, this technique is also called *offline trace checking* or *post-mortem analysis*, in order to distinguish it from online techniques that check the correctness of a system *while* it is executing [FHR13], possibly by processing a stream of events [DSS+05, Hal16]. The two main aspects characterizing trace checking approaches are: 1) the type of properties to verify (as well as the specification language used to express them), and 2) the actual checking procedure to use.

Chaima Boufaied, Domenico Bianculli, Lionel Briand. *A Model-driven Approach to Trace Checking of Temporal Properties with Aggregations*. Licensed under Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0). In Journal of Object Technology, vol. 18, no. 2, 2019, pages 15:1–21. doi:10.5381/jot.2019.18.2.a15

The system properties verified in the context of trace checking can be specified using different specification languages and formalisms, such as a temporal logic (e.g., linear temporal logic, metric temporal logic, signal temporal logic), regular expressions, state machines, or a combination of them [BFFR18], possibly using domain-specific languages (DSLs). In this paper, we consider a specific class of temporal properties: *the ones containing operators that aggregate events*. An example of such a property is: "the average number of client requests per hour computed over the daily business hours (from 7.30AM to 7PM) should be less than 10000", where the operator "average" is used to aggregate the number of events of type "client request" over one-hour intervals, in an observation time window ranging "from 7.30AM to 7PM". This class of temporal properties was identified in a field study [BGPS12], which analyzed more than 900 requirements specifications written in the context of service-based applications, extracted from research papers and industrial data (in the domain of banking service provisioning). More specifically, the study identified a new class of property specification patterns (inspired by Dwyer et al.'s seminal work [DAC99]) called *service provisioning patterns*, which matched the majority of the requirements specifications stated in industrial settings. More than 80% of the industrial specifications analyzed in the study could be written as temporal properties using aggregation operators such as average, maximum, and average response time. To the best of our knowledge, the only specification language supporting the service provisioning patterns identified in [BGPS12] is *SOLOIST* [BGS13], a language based on first-order metric temporal logic extended with aggregating modalities.

In terms of checking procedure, in this paper we consider a *model-driven trace checking* approach [DBB17a]. Such an approach consists in *reducing* the problem of checking a property ρ over an execution trace λ to the problem of evaluating an OCL (Object Constraint Language) constraint (semantically equivalent to ρ) on an instance (equivalent to λ) of a meta-model of the trace. Model-driven trace checking has been proposed in the literature [DBB17a] *as a viable solution to the trace checking problem, to be adopted in software development contexts that rely on a model-driven engineering (MDE)*, a common practice in many domains [BCW17]. TemPsy-Check [DBB17b] is a state-of-the-art tool implementing model-driven trace checking; it has been shown [DBB17a] to be highly scalable with respect to the length of the trace, exhibiting performance that is comparable to (and in some cases better than) state-of-the-art alternative technologies based on temporal logic. The properties to verify with TemPsy-Check are expressed in *TemPsy* (Temporal Property made easy) [DBB17a], a pattern-based DSL for the specification of temporal properties. This language is based on the catalogue of property specification patterns by Dwyer et al. [DAC99], with new constructs derived from a field study performed in the domain of business processes for eGovernment. *TemPsy* is suitable for adoption by practitioners, since it is a high-level specification language that does not require a strong theoretical and mathematical background. Furthermore, *TemPsy* is pattern-based and inherits the benefits of pattern-based languages: for example, a recent empirical study [CZss] has shown that pattern-based temporal property specifications are easier to understand than specifications written using Linear Temporal Logic and the Event Processing Language.

The work proposed in this paper is motivated by two observations. On one hand, though requirements specifications based on temporal properties with aggregation operators (i.e., those based on the "service provisioning" specification patterns) are common, the support provided in terms of verification (more specifically, trace check-

ing) of such properties is limited. Indeed, the two non-distributed[1] trace checking algorithms for *SOLOIST* proposed in the literature [BGKSP14, BBG+14] do not scale well in terms of the length of the trace [BGK14]. On the other hand, there is TemPsy-Check, a scalable and effective model-driven trace checking solution, which only supports temporal properties based on Dwyer et al.'s specification patterns.

The goal of this paper is *to provide a solution for scalable model-driven trace checking of temporal properties with aggregation operators*. The main idea is to bridge the gap between property specifications based on service provisioning patterns and model-driven trace checking. More specifically, we extend the approach presented in [DBB17a] by proposing: 1) an extension of the *TemPsy* language called *TemPsy-AG*, which supports the most used service provisioning patterns identified in the study in [BGPS12]; 2) an extension of the *TemPsy* trace checking procedure, realized through an optimized mapping into OCL constraints (on a meta-model of execution traces) of the new types of properties included in *TemPsy-AG*.

We have implemented our approach in TemPsy-Check-AG, a prototypical extension of TemPsy-Check. We evaluated its scalability (in terms of execution time) with respect to the length of the trace and other parameters used in the specification of *TemPsy-AG* properties; we also compared its performance with respect to *SOLOIST*-translator, the state-of-the-art tool for (non-distributed) trace checking of *SOLOIST* specifications [BGKSP14, BBG+14]. The results show that TemPsy-Check-AG scales linearly with respect to the parameters considered in the analysis; in particular its execution time for processing a large trace with one million events ranges between 6250 ms and 15339 ms, depending on the aggregation operator used in the property. Furthermore, it can deal with much larger traces than *SOLOIST-Translator*, which could only handle traces up to 1500 events.

To summarize, the two main contributions of the paper are: i) a model-driven approach for trace checking of temporal properties with aggregation operators; ii) an evaluation of the scalability of such an approach when implemented in the TemPsy-Check-AG tool, and the comparison with a state-of-the-art alternative technology. In addition, this paper can be seen as a *successful case study* on the feasibility and viability of extending model-driven trace checking [DBB17a], i.e., a verification technique enabled by MDE technologies, with support for a larger class of properties, while retaining acceptable performance from a practical standpoint.

The rest of the paper is structured as follows. Section 2 provides some background concepts on *TemPsy*, model-driven trace checking, and service provisioning patterns. Section 3 presents *TemPsy-AG*. Section 4 illustrates our approach for model-driven checking of temporal properties based on service provisioning patterns. Section 5 reports on the evaluation of the scalability of our implementation. Section 6 discusses related work. Section 7 concludes the paper, giving directions for future work.

2 Background

In this section, we first give a short overview of the *TemPsy* language and of the corresponding model-driven trace checking procedure realized by TemPsy-Check; afterwards, we present the specification patterns for service provisioning.

[1] The issue of distributed trace checking using Big Data technologies is out of the scope of this paper (which restricts itself to the non-distributed case) and is left for future work; a distributed algorithm for *SOLOIST* has been presented in [BGK14].

⟨*TemPsyExpression*⟩ ::= [‘temporal’ ⟨*Id*⟩ ‘:’]
  ⟨*Scope*⟩ ⟨*Pattern*⟩
⟨*Scope*⟩ ::= ‘globally’
  | ‘before’ ⟨*Boundary1*⟩
  | ‘after’ ⟨*Boundary1*⟩
  | ‘between’ ⟨*Boundary2*⟩ ‘and’ ⟨*Boundary2*⟩
  | ‘after’ ⟨*Boundary2*⟩ ‘until’ ⟨*Boundary2*⟩
⟨*Pattern*⟩ ::= ‘always’ ⟨*Event*⟩
  | ‘eventually’ ⟨*RepeatableEventExp*⟩
  | ‘never’ [‘exactly’ ⟨*Int*⟩] ⟨*Event*⟩
  | ⟨*EventChainExp*⟩ ‘preceding’
  [⟨*TimeDistanceExp*⟩] ⟨*EventChainExp*⟩
  | ⟨*EventChainExp*⟩ ‘responding’
  [⟨*TimeDistanceExp*⟩] ⟨*EventChainExp*⟩
⟨*Boundary1*⟩ ::= [⟨*Int*⟩] ⟨*Event*⟩ [⟨*TimeDistanceExp*⟩]
⟨*Boundary2*⟩ ::= [⟨*Int*⟩] ⟨*Event*⟩ [‘at least’ ⟨*Int*⟩ ‘tu’]
⟨*EventChainExp*⟩ ::= ⟨*Event*⟩
  (‘,’ [‘#’ ⟨*TimeDistanceExp*⟩] ⟨*Event*⟩)*
⟨*TimeDistanceExp*⟩ ::= ⟨*ComparingOp*⟩ ⟨*Int*⟩ ‘tu’
⟨*RepeatableEventExp*⟩ ::= [⟨*ComparingOp*⟩ ⟨*Int*⟩] ⟨*Event*⟩
⟨*ComparingOp*⟩ ::= ‘at least’ | ‘at most’ | ‘exactly’
⟨*Event*⟩ ::= ⟨*Id*⟩

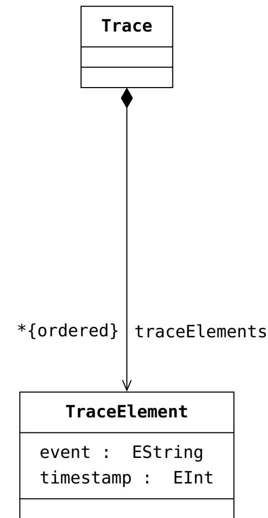

Figure 1 – Syntax (fragment) of *TemPsy* (left) and meta-model for execution traces (right)

2.1 The *TemPsy* language

TemPsy (Temporal Property made easy) [DBB17a] is a pattern-based, domain-specific language for the specification of temporal properties. It has been designed based on the catalogue of property specification patterns by Dwyer et al. [DAC99], with new constructs derived from a field study performed in the domain of business processes for eGovernment.

The main elements of the syntax of *TemPsy* are shown on the left of Figure 1: terminals are enclosed in single quotes, non-terminals in angle brackets, optional elements in brackets; character ‘*’ indicates zero or more occurrences of an element; ⟨*event*⟩s are denoted by alphanumeric strings. Temporal properties in *TemPsy* are represented through the concept of ⟨*TemPsyExpression*⟩, which is composed of a *scope* and a *pattern*: the latter represents a high-level abstraction of a formal specification while the former indicates the portion(s) of a system execution in which a certain pattern should hold. *TemPsy* supports all the patterns (“absence”, “universality”, “existence”, “bounded existence”, “precedence”, “response”, “precedence chain”, “response chain”) and scopes (“globally”, “before”, “after”, “between-and”, “after-until”) introduced in [DAC99]; patterns and scopes are denoted by an intuitive syntax.

The new constructs added in *TemPsy* on top of the original patterns definitions in [DAC99] include: the possibility, in the definition of a scope boundary, of i) referring to a specific occurrence of an event, and of ii) indicating a distance from the scope boundary; iii) the support for indicating a time distance between occurrences in the *precedence* and *response* patterns (as well as their chain versions); iv) additional variants for the bounded existence and absence patterns. Notice that 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 [KC05]).

For example, the property "Event A shall happen at least 2 time units after the fifth occurrence of event X" is expressed as "**`after`** `5 X` **`at least`** `2` **`tu eventually`** `A`".

2.2 Model-driven trace checking with *TemPsy*-Check

Model-driven trace checking [DBB17a] is an approach that *reduces* the problem of checking a temporal property ρ over an execution trace λ to the problem of evaluating an OCL constraint (semantically equivalent to ρ) on an instance (equivalent to λ) of a meta-model of the trace. This reduction enables the use of standard constraint checking technology to perform trace checking; standard OCL checkers, such as Eclipse OCL[2], can be used to evaluate OCL constraints on model instances in a practical and scalable way. This trace checking technique has been proposed for adoption in contexts that rely on a model-driven development process, in which solutions must be engineered by using standard MDE technologies that are already in place in the targeted development environment.

In the case of *TemPsy*, the model-driven trace checking approach has been implemented in the TemPsy-Check tool [DBB17b]. This tool relies on an optimized mapping of *TemPsy* properties into OCL constraints on the meta-model of execution traces depicted in the UML class diagram shown on the right of Figure 1. The meta-model contains a `Trace` class composed of a sequence of `TraceElements` accessed through the association `traceElements`. Each `TraceElement` contains an attribute `event` of type `string`, which represents an event recorded in the trace, and an attribute `timestamp` of type `integer`, which indicates the time at which the event occurred.

In a nutshell, TemPsy-Check works as follows: given a trace and a *TemPsy* property (represented by a scope s and a pattern p), the tool evaluates an OCL invariant defined based on the type of s and p. This evaluation conceptually corresponds to applying the semantics of pattern p on a set of sub-traces; the latter is determined by the semantics of scope s. The output of the invariant evaluation is then returned as Boolean verdict of the trace checking procedure.

2.3 Specification Patterns for Service Provisioning

Specification patterns for service provisioning were identified in a field study [BGPS12], which analyzed more than 900 requirements specifications written in the context of service-based applications, extracted from research papers and industrial data. The study classified the requirements specifications according to four systems of property specification patterns: three of them were already defined in the literature [DAC99, KC05, GL06] whereas the fourth class included patterns that emerged during the study. Indeed, the majority of the requirements specifications stated in industrial settings (in the domain of banking service provisioning) could be expressed through this new set of patterns, which consists of: "average response time" (S1), "counting the number of events" (S2), "average number of events" (S3), "maximum number of events" (S4), "absolute time" (S5), "unbounded elapsed time" (S6), and "data awareness" (S7). Among these, patterns S1-S3-S4 were used in almost 82% of the specifications.

[2] https://projects.eclipse.org/projects/modeling.mdt.ocl

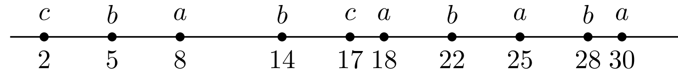

Figure 2 – Sample trace

In the following, we provide an explanation of patterns S1-S3-S4 (which are the ones considered in this paper), based on the semantics adopted by the *SOLOIST* language [BGS13], which is a temporal logic tailored to the specification of properties based on the service provisioning patterns. Figure 2 depicts a sample trace (where letters in the upper part of the timeline correspond to events, and numbers in the lower part of the timeline indicate time-stamps in seconds) that will be used to explain the patterns.

***Average response time* (S1).** This pattern (also known as *average elapsed time*) is a variant of the bounded response time proposed in [KC05], in which the distance between pairs of events (i.e., the response time) is aggregated over a time window of length K using the average operator. It can be used to express a property like "P1: *The average distance between events a and b in the last 20 seconds should be less than 3*", where $K = 20$. The evaluation of this property, when done in the position corresponding to the last element of the trace in Figure 2 (with timestamp $\tau_{end} = 30$), will consider the time window of length K that includes the events with a time-stamp τ such that $\tau_{end} - K < \tau \leq \tau_{end}$: $(b, 14)$, $(c, 17)$, $(a, 18)$, $(b, 22)$, $(a, 25)$, $(b, 28)$, $(a, 30)$. The average distance is then computed by summing the differences between the time-stamps of each pair[3] of events (a, b) and dividing the result by the number of the selected events pairs (2 in the example). In this case the average elapsed time is $\frac{(22-18)+(28-25)}{2} = 3.5$, which is greater than the bound 3, thus violating the property.

***Average number of events* (S3).** This pattern aggregates (using the average operator) the number of events that occurred in an observation interval h within a time window K. It can be used to express a property like "P3: *Within the last 20 seconds, in each 6-second interval, the average number of occurrences of event a should be less than 3*", where $K = 20, h = 6$. The evaluation of this property, when done in the position corresponding to the last element of the trace in Figure 2 (with timestamp $\tau_{end} = 30$), will consider the time window that includes the events with a time-stamp τ such that $\tau_{end} - \lfloor \frac{K}{h} \rfloor h < \tau \leq \tau_{end}$. Notice that the left boundary of the time window is determined by taking into account the possibility that K may not be an exact multiple of h; if this is the case, the tail interval (with length shorter than the observation interval) is discarded. This time window is then split in $\lfloor \frac{K}{h} \rfloor$ adjacent, non-overlapping observation intervals (open to the left and closed to the right) of length $h = 6$; in the example, we have $\lfloor \frac{20}{6} \rfloor = 3$ observation intervals, delimited by the following time-stamp boundaries: $(12, 18]$, $(18, 24]$, and $(24, 30]$. The average number of occurrences is then computed by summing all the occurrences of event a in each observation interval, and dividing the result by the number of observation intervals. In the example, the average number of occurrences of a is $\frac{1+0+2}{3} = 1$, which

[3] Based on the semantics in [BGS13], the event $(a, 30)$ is ignored for computing the (average) distance, since it is not matched by a corresponding b event within the selected time window; a similar reasoning applies to event $(b, 14)$, which does not follow any event a in the selected time window. Furthermore, as proposed in [BGS13], we require that between two occurrences of event a (respectively, event b), there is an occurrence of event b (respectively, event a); because of this assumption, fragments of traces of the form *aabb* will be collapsed into *ab* in a pre-processing step.

is less than the bound 3: the property is satisfied.

***Maximum number of events* (S4).** This pattern is a variant of the previous one, in which the events are aggregated using the maximum operator. It can be used to express a property like "P4: *Within the last 20 seconds, in each 6-second interval, the maximum number of occurrences of event a should be less than 3*"; also in this case, $K = 20$ is the time window considered for the aggregation, and $h = 6$ is the observation interval. Differently from the case of pattern S3, the semantics of this pattern (as defined in [BGS13]) takes also into account the events occurring in the tail interval, even if its length is shorter than the one of the observation interval h. The evaluation of property P4, when done in the position corresponding to the last element of the trace in Figure 2, will thus consider $\lceil \frac{20}{6} \rceil = 4$ observation intervals, delimited by the following time-stamp boundaries: $(10, 12]$, $(12, 18]$, $(18, 24]$, and $(24, 30]$. The application of the maximum operator will yield the value 2, which is less than the bound 3: the property is satisfied.

3 Specifying temporal properties with aggregation operators through *TemPsy-AG*

As a preliminary step towards our model-driven approach for trace checking of temporal properties with aggregations, we extended the *TemPsy* language to support the most used service provisioning patterns (i.e., S1, S3, and S4); the new version of the language is called *TemPsy-AG*. We modified the syntax of *TemPsy* by adding new rules corresponding to the constructs needed for the new patterns; the main additions to the grammar are shown in Figure 3. More specifically, *TemPsy-AG* sports three new, intuitive keywords (`avgRT`, `average`, `maximum`) indicating, respectively, patterns S1, S3, and S4. In all these patterns, the time window used for aggregation is denoted through the `within` keyword; the observation interval for patterns S3 and S4 is represented with the `every` keyword; the bound is expressed through the non-terminal ⟨*Bound*⟩, supporting the usual relational operators.

The example properties presented in section 2.3 can be written in *TemPsy-AG* as follows (assuming a *globally* scope):

- *P1*: `globally avgRT(a,b) within 20 tu < 3`
- *P3*: `globally average a within 20 tu every 6 tu < 3`
- *P4*: `globally maximum a within 20 tu every 6 tu < 3`

In the same spirit as *TemPsy*, by design *TemPsy-AG* does not aim at being as expressive as a full-fledged temporal logic. Instead, its goal is to make as easy as

```
⟨Pattern⟩ ::= ...
   | 'avgRT' '('⟨Event⟩ ',' ⟨Event⟩')' 'within' ⟨Int⟩ 'tu' ⟨Bound⟩
   | 'average' ⟨Event⟩ 'within'⟨Int⟩ 'tu' 'every'⟨Int⟩ 'tu' ⟨Bound⟩
   | 'maximum' ⟨Event⟩ 'within'⟨Int⟩ 'tu' 'every'⟨Int⟩ 'tu' ⟨Bound⟩
     ...
⟨Bound⟩ ::= ('>' | '>=' | '<' | '<=' | '==' | '!=') ⟨Int⟩
```

Figure 3 – Syntactic extension included in *TemPsy-AG*

```
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   [...code related to the other patterns omitted...]
7   if pattern.type = TemPsy::MAX then
8     subtraces->forAll(subtrace | checkPatternMAX(subtrace, pattern))
9    else if pattern.type = TemPsy::AVG then
10    subtraces->forAll(subtrace | checkPatternAVG(subtrace, pattern))
11   else if pattern.type = TemPsy::AVGRT then
12    subtraces->forAll(subtrace | checkPatternAVGRT(subtrace, pattern)
13   endif endif endif
14  else if scope.type = TemPsy::BEFORE then [...])
```

Figure 4 – OCL invariant for checking *TemPsy-AG* properties on a trace

possible the specification of common types of temporal properties through a pattern-based language.

The formal definition of the semantics of *TemPsy-AG* (not included here for space reasons) extends the one of *TemPsy* (available in [Dou16]), and is based on the concept of temporal linear traces. The semantics of the new three operators added in *TemPsy-AG* largely mirrors the formalization of the corresponding service provisioning patterns provided in [BGS13]. The main difference from the definitions in [BGS13] is that the operators corresponding to the service provisioning patterns are always evaluated in correspondence of the last element of a (sub-)trace; in other words, the time window defined by the parameter K is always computed with respect to the timestamp of the last element of the (sub-)trace (as done in section 2.3).

4 Model-driven trace checking of *TemPsy-AG* properties

Our approach for model-driven trace checking of temporal properties with aggregation operators is based on the existing trace checking procedure available in TemPsy-Check [DBB17b, DBB17a]. In this section we present the extension of this procedure to support the new types of properties included in *TemPsy-AG*.

Our main contribution is the operationalization, in OCL, of the semantics of the service provisioning patterns included in *TemPsy-AG*. This is a challenging task since this mapping has to be optimized, based on the structure of the properties to check, in order to achieve better performance.

As mentioned in section 2.2, the idea at the basis of model-driven trace checking, given as input a *TemPsy-AG* property represented by a scope s and a pattern p, is to evaluate an OCL invariant defined based on the type of s and p. This evaluation conceptually corresponds to applying the semantics of pattern p on the set of sub-traces that is determined by the semantics of scope s.

We extend the definition of this invariant to support the aggregation operators available in *TemPsy-AG*; a snippet of its OCL pseudocode is shown in Figure 4. The body of the invariant expression is a multi-way branch, which selects a certain branch based on the specific scope type used within the property (line 4 shows the case for the

"globally" scope). In each branch, after determining the collection of sub-traces (as determined by the scope semantics) with a call to a function of the form `applyScope*` (as in line 5, invoking `applyScopeGlobally`), there is another multi-way branch, which selects a certain branch based on the specific pattern type used within the property. Lines 7–13 show the branches corresponding to the three new aggregation operators of *TemPsy-AG*. In each branch there is a function of the form `checkPattern*` that checks whether the pattern used in the property holds on each sub-trace. The core of our extension lies in the definition of three new OCL functions—`checkPatternAVGRT` (for pattern S1, "average response time"), `checkPatternAVG` (for pattern S3, "average number of events"), and `checkPatternMAX` (for pattern S4, "maximum number of events")—that contain the operational definition (in OCL) of the semantics of each pattern. The rest of the invariant definition (e.g., returning the verdict) is the same as in the original version [DBB17a].

In the rest of this section we illustrate the definition of the three new `checkPattern*` aforementioned functions, corresponding to the S1, S3, S4 service provisioning patterns; to ease readability, the algorithms are written using OCL pseudocode.

4.1 Checking the "average response time" pattern

Function `checkPatternAVGRT`, whose pseudocode is shown in Algorithm 1, takes as input a sub-trace and the parameters of an object representing an *average response time* pattern in *TemPsy-AG*: the pair of events (a, b), the length of the time window K, and a bound expressed with a relational operator $\bowtie$ and a numeric constant n. The function returns a Boolean value indicating whether the pattern holds on the input sub-trace, i.e., whether the cumulative sum (over all pairs of events (a, b) in the time window) of the time distance between each occurrence of event b and the corresponding occurrence of event a, divided by the number of (a, b) events pairs in the time window, satisfies the given bound.

The algorithm uses four auxiliary variables: *accDist* represents the cumulative sum (over all pairs of events (a, b)) of the time distance between each occurrence of event b and the corresponding occurrence of event a; *numPairs* is a counter keeping track of the number of (a, b) events pairs found; *inPair* is a Boolean flag that is true when the event a has been seen and the corresponding event b has not been seen yet; *lastSeenTS* is the timestamp of the last-seen occurrence of event a.

First (line 2), the function determines the timestamps corresponding to the left and right boundaries of the sub-trace to consider. The right boundary RB is the timestamp of the last element of the sub-trace, whereas the left boundary LB is determined by the value of the time window and is equal to $RB - K$; notice that parameter K is assumed to be greater than the sub-trace length.

The central block of the function is a loop (lines 3–11) that iterates over all elements of the input sub-trace whose timestamp is comprised between the left and the right boundaries of the time window[4]. For each element, we check whether the corresponding event is a match for either a or b. If we match an occurrence of event a, we set the flag *inPair* to `true` and save the corresponding timestamp in variable *lastSeenTS*. Notice that, as discussed in the footnote on page 6, we assume that between two occurrences of event a there is an occurrence of event b. If we match an occurrence of event b, if the flag *inPair* is `true` it means that this event is the one corresponding to the last occurrence of event a previously matched. In this case, we

[4]We consider the time window interval closed to the right, open to the left.

Algorithm 1: checkPatternAVGRT

```
Input: a trace segment subtrace and the parameters of an instance of the
       average response time pattern of the form avgRT(a,b) within K tu ⋈ n:
Output: true if the pattern holds on subtrace; false otherwise
1  accDist ← 0, numPairs ← 0
   inPair ← false, lastSeenTS ← null
2  RB ← subtrace.last().timestamp, LB ← RB − k
3  foreach elem ∈ subtrace such that
   LB < elem.timestamp ≤ RB do
4      if elem.event = a then
5          inPair ← true
6          lastSeenTS ← elem.timestamp
7      else if elem.event = b then
8          if inPair = true then
9              accDist ← accDist + (elem.timestamp − lastSeenTS)
10             numPairs ← numPairs + 1
11             inPair ← false
12 return evalBound (accDist/numPairs, ⋈, n)
```

compute the distance between this element and the timestamp of the last-seen occurrence of event a, update the value of *accDist* accordingly, increase by 1 the value of the *numPairs* counter, and reset the value of *inPair* to `false`. By construction, following the informal semantics of the pattern presented in section 2.3, the algorithm will ignore the occurrences of event a that are not matched by a corresponding b event within the time window, as well as the occurrences of event b that do not follow any event a in the time window.

The average response time is then computed by dividing the value of the variable *accDist* (i.e., the cumulative time distance) by the value of variable *numPairs* (i.e., the number of matched pairs). This value is passed to function *evalBound*, together with the value of parameters $\bowtie$ and n. This function evaluates the bound stated in the property, according to the relational operator $\bowtie$ and the numeric constant n; the result is then returned by the algorithm, representing the Boolean verdict of the trace checking procedure.

4.2 Checking the "average number of events" pattern

Function `checkPatternAVG`, whose pseudocode is shown in Algorithm 2, takes as input a sub-trace and the parameters of an object representing an *average number of events* pattern in *TemPsy-AG*: the event a, the length of the time window K, the length of the observation interval h, a bound expressed with a relational operator $\bowtie$, and a numeric constant n. The function returns a Boolean value indicating whether the pattern holds on the input sub-trace, i.e., whether the average number of occurrences of event a, aggregated over the observation intervals that are included in the time window, satisfies the given bound.

First, the algorithm computes the temporal boundaries of the sub-trace to consider: the right boundary RB is the timestamp of the last element of the sub-trace; the left boundary LB depends on the value of the time window and on the value of

Algorithm 2: checkPatternAVG

Input: a trace segment *subtrace* and the parameters of an instance of the *average number of events* pattern of the form **average** a **within** K **tu** **every** h **tu** $\bowtie n$:

Output: `true` if the pattern holds on *subtrace*; `false` otherwise

1 $RB \leftarrow subtrace$.`last()`$.timestamp$, $LB \leftarrow RB - \lfloor \frac{K}{h} \rfloor$

2 $totalOccurrences \leftarrow count(subtrace,(LB,RB),a)$

3 $numIntervals \leftarrow \lfloor \frac{K}{h} \rfloor$

4 **return** $evalBound\left(\frac{totalOccurrences}{numIntervals}, \bowtie, n\right)$

the observation interval and is equal to $RB - \lfloor \frac{K}{h} \rfloor$.

Algorithm 3: checkPatternMAX

Input: a trace segment *subtrace* and the parameters of an instance of the *maximum number of events* pattern of the form
maximum a **within** K **tu every** h **tu** $\bowtie n$

Output: `true` if the pattern holds on *subtrace*; `false` otherwise

```
1 RB ← subtrace.last().timestamp
2 intervals ← getIntervals(RB,K,h)
3 foreach itv ∈ intervals do
4     numOccurrences.append(count(subtrace,itv,a))
5 return evalBound (max (numOccurrences) ,⋈, n)
```

Then the algorithm computes two values:

- the number of occurrences of event a occurring in the interval $(LB, RB]$, computed using the auxiliary function *count* and stored in variable *totalOccurrences*;
- the number of observation intervals over which to compute the aggregate value, which is equal to $\lfloor \frac{K}{h} \rfloor$ according to the semantics of the pattern (see section 2.3); this value is stored in variable *numIntervals*.

The average number of events is then computed by dividing the value of *totalOccurrences* by the value of *numIntervals*. The resulting value is passed to function *evalBound*, together with the value of parameters $\bowtie$ and n, to evaluate the bound stated in the property and determine the verdict of the trace checking procedure.

4.3 Checking the "maximum number of events" pattern

Function `checkPatternMAX`, whose pseudocode is shown in Algorithm 3, takes as input a sub-trace and the parameters of an object representing a *maximum number of events* pattern in *TemPsy-AG*: the event a, the length of the time window K, the length of the observation interval h, a bound expressed with a relational operator $\bowtie$, and a numeric constant n. The function returns a Boolean value indicating whether the pattern holds on the input sub-trace, i.e., whether the maximum number of occurrences of event a, aggregated over the observation intervals that are included in the time window, satisfies the given bound.

First, the algorithm computes in variable RB the temporal right boundary of the trace, i.e., the timestamp of the last element. Then it determines—by calling the auxiliary function *getIntervals*—the left and right temporal boundaries of each observation interval in the sub-trace, based on the values of RB, K, and h. Function *getIntervals* will return a list with $\lceil \frac{K}{h} \rceil$ intervals; these intervals are open to the left and closed to the right. In this list, stored in variable *intervals*, the first $\lceil \frac{K}{h} \rceil - 1$ intervals have length h and have the form $(RB - (m+1)h, RB - mh]$ for $0 \leq m \leq \left(\lfloor \frac{K}{h} \rfloor - 1\right) h$; the last interval (i.e., the left-most on the timeline) in the list will be $\left(\max\left(RB - K, RB - \lceil \frac{K}{h} \rceil\right), RB - \lfloor \frac{K}{h} \rfloor h\right]$, to take into account the possibility of having a tail interval shorter than h, according to the semantics of the pattern described in section 2.3.

Afterwards, the loop at lines 3–4 computes (through the auxiliary function *count*), for each interval *itv* in the list *intervals*, the number of occurrences of event a in *itv* and stores this value in the set *numOccurrences*.

The maximum number of events is then determined by computing the maximum

value over the set *numOccurrences*. This value is passed to function *evalBound*, together with the value of parameters $\bowtie$ and n, to evaluate the bound stated in the property and determine the verdict of the trace checking procedure.

4.4 Tool implementation

We have implemented our approach in TemPsy-Check-AG, as an extension of the publicly available TemPsy-Check [DBB17b] tool. The extension includes the OCL code to deal with the service provisioning patterns; we have also extended the *TemPsy* DSL editor to support the new expressions in the *TemPsy-AG* language.

Our extension uses the same toolchain as TemPsy-Check: it takes as input a trace in CSV format and a text file following the DSL syntax, with the properties to check on the trace; the evaluation of the OCL constraints corresponding to the input properties to check is performed using the OCL checker included in the Eclipse OCL distribution.

5 Evaluation

We evaluated the scalability of our approach—in terms of the execution time—with respect to the length of the trace and other parameters used in the specification of *TemPsy-AG* properties. We also compared the performance of our approach with *SOLOIST-Translator*, the state-of-the-art tool for (non-distributed) trace checking of *SOLOIST* specifications [BGKSP14, BBG+14]. More specifically, we evaluated our approach implemented in TemPsy-Check-AG by answering the following research questions:

RQ1: How does TemPsy-AG *scale with respect to the trace length when checking properties expressed using the three main service provisioning patterns (S1, S3, S4)?* (section 5.2.1)

RQ2: How does TemPsy-AG *scale with respect to the number of observation intervals induced by the values of the parameters K and h, when checking properties expressed using patterns S3 and S4?* (section 5.2.2)

RQ3: How does TemPsy-AG *fare with respect to* SOLOIST-Translator*, a state-of-the-art tool for checking properties expressed using the three main service provisioning patterns (S1, S3, S4)?* (section 5.2.3)

5.1 Evaluation Settings

5.1.1 Temporal Properties

We used the three following property templates to answer all three research questions, one for each type of service provisioning pattern:

- *P1*: **globally avgRT**(a,b) **within** K **tu** < 5 (for pattern S1)
- *P3*: **globally average** a **within** K **tu every** h **tu** < 5 (for pattern S3)
- *P4*: **globally maximum** a **within** K **tu every** h **tu** < 5 (for pattern S4)

where a and b are event names and K, h are parameters that are varied according to the evaluation methodology (described below). Notice that all properties are expressed using the "globally" scope: we made this choice following the evaluation methodology proposed in existing work on model-driven trace checking [DBB17a]. Indeed, properties with the "globally" scope are the 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 throughout the entire length of the trace.

5.1.2 Trace Generation Strategy

Following the evaluation guidelines proposed in existing work [BBG+16, DBB17a] on trace checking, we used *synthesized* traces for the evaluation. The use of synthesized traces over real ones allows us to systematically control the factors (e.g., the trace length, the number of intervals) that are relevant for our research questions, while setting other factors randomly, to avoid any bias.

We extended the trace generator program included in the TemPsy-Check distribution with new generator strategies specific to the service provisioning patterns. The generator program takes as input a *TemPsy-AG* 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. The position and the order of events are generated randomly 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 deemed not relevant for the evaluation of the property are filled with a dummy event, so that *the number of events in the trace is equal to the parameter K* used in patterns S1, S3, S4. In other words, between two adjacent events in the trace we assume a time difference of 1 time unit, possibly indicated by the presence of a dummy event. Given the semantics of the service provisioning patterns and taking into account their formalization in *TemPsy-AG* (see section 3), this case corresponds to the worst-case scenario (from a scalability point of view), in which the time window over which the properties with aggregations are evaluated includes all the elements of the trace. Below we sketch the trace generation strategies for the three new patterns.

***Average response time* (P1).** For a given value of the parameter K we generate a trace of length K, containing X pairs of events (a, b), where X is a random value between 2 and $\frac{K}{2}$. We require that these pairs are distributed over the trace such that the average time distance between the individual occurrences of events a and the corresponding occurrences of b satisfies the bound indicated in the property. We use the Z3 constraint solver [DMB08] to get the value of the X distances (one for each events pair) that satisfy the property bound. Then we randomly allot the pairs of (a, b) events over the trace according to a uniform distribution, while maintaining for each pair the distance determined by the solver.

***Average number of events* (P3).** For a given value of the parameters K and h, we generate a trace of length K; the number of observation intervals I is computed as $I = \lfloor \frac{K}{h} \rfloor$. We then need to determine the number of occurrences of event a in each of these I intervals, such that their distribution on the trace satisfies the property bound. We use the Z3 constraint solver to find an assignment for the I variables that represent the number of event occurrences in each interval. Finally, within each interval, we randomly generate (with a uniform distribution on the range induced by h) the required number of occurrences of events a in that interval.

***Maximum number of events* (P4).** For a given value of the parameters K and h, we generate a trace of length K; the number of observation intervals I is

computed as $I = \lceil \frac{K}{h} \rceil$. We then need to determine the number of occurrences of event a in each of these I intervals, such that their distribution on the trace satisfies the property bound. For example, if the bound used in the property is "`< n`", in each observation interval we will generate occurrences of event a with a uniform distribution on the range $[0, \mathsf{n}]$.

5.1.3 Computer Settings

The results reported in this section have been measured using the Unix `time` program on a desktop computer with a 2.5 GHz Intel Core i7 CPU and 16 GB of memory, running Eclipse DSL Tools v. 4.6.2 (Neon Milestone 2), JavaSE-1.7 (Java SE v. 1.8.0_121, Java HotSpot (TM) 64-Bit Server VM v. 25.121-b13, mixed mode), Eclipse OCL v. 6.1.2, and *SOLOIST-Translator* (most recent version, commit `65684d1`). All measurements reported correspond to the average value over 5 runs of the trace checking procedure (on the same trace, for the same property).

5.2 Evaluation Results

5.2.1 Scalability with respect the to trace length

Methodology. To answer RQ1, we ran TemPsy-Check-AG on traces of different length (i.e., value of the parameter K in the input property), ranging from 100K to 1M in steps of 100K; on each trace, we checked the three properties shown above. For P3 and P4 properties, we varied h so that the number of intervals was fixed to 10.

Results. The execution time measured for our approach is shown in Figure 5a. Overall, the execution time varies from 2218 ms, for checking property P1 (with the "average response time" pattern) on a 100K trace, to 15339 ms, for checking property P3 (with the "average number of events" pattern) on a 1M trace. We also notice that checking properties with the "average/maximum number of events" patterns requires longer than checking properties with the "average response time" pattern.

The answer to RQ1 is that our approach *scales linearly* with respect to the length of the trace for all three types of service provisioning patterns.

5.2.2 Scalability with respect to the number of observation intervals

Methodology. To address RQ2, we generated traces with length varying from 100K to 500K in steps of 100K; for each of these trace lengths, we considered 10 different values for the number of observation intervals (used in the context of patterns S3 and S4), ranging from 10 to 100 in steps of 10. In practice, to vary the number of observation intervals, we varied the value of parameter h in properties P3 and P4 such that, when combined with the value of parameter K, it yields the desired value for the number of observation intervals. For example, in the case of a property with the "average number of events" pattern, the value of h for obtaining 30 intervals on a 100K long trace is $h = \lfloor \frac{100000}{30} \rfloor = 3333$. Notice that in the case of the "maximum number of events" pattern, when $K \mod h \neq 0$, the tail interval will have a length shorter than h and there is a range of values for h that yield the same number of observation intervals; in this case, we chose the lowest value for h to get the largest possible tail interval. We executed TemPsy-Check-AG on all the generated traces, for the different values of K and h, to check properties P3 and P4.

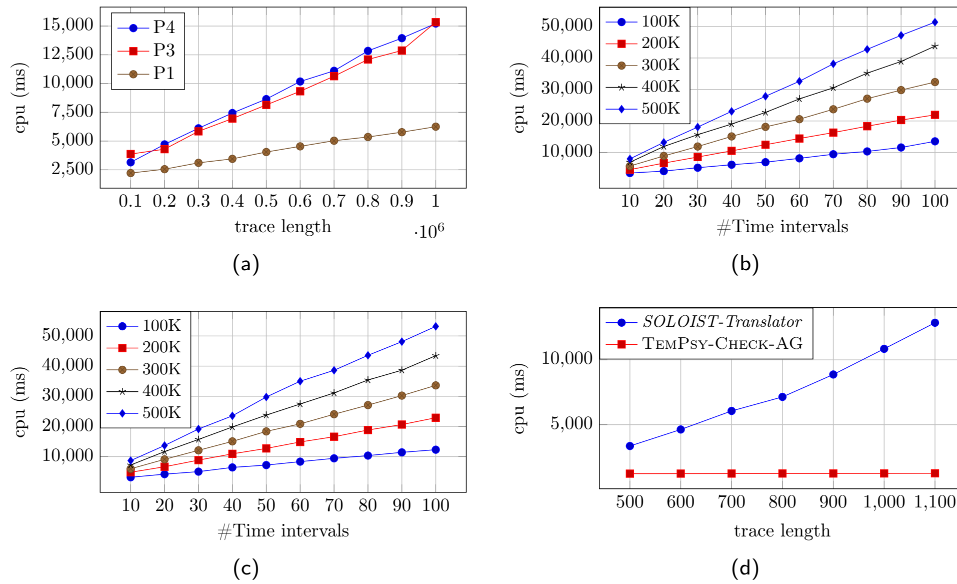

Figure 5 – Scalability in terms of execution time with respect to the trace length (a); scalability in terms of the number of observation intervals for P3 properties (b) and P4 properties (c); comparison between the execution time of TemPsy-Check-AG and of *SOLOIST-Translator* for checking P1 properties (d)

Results. The execution time for checking properties with pattern S3 "average number of events" and with pattern S4 "maximum number of events" is shown in Figure 5b and Figure 5c, respectively. Overall, the execution time ranges from 3138 ms, for checking a property with pattern S3 on a 100K long trace with 10 time intervals, to 53183 ms, for checking a property with pattern S3 on a 500K long trace with 100 time intervals. In line with the results discussed for RQ1, the execution time for checking properties P3 and P4 is similar, i.e., checking patterns S3 and S4 has a similar cost.

The answer to RQ2 is that TemPsy-Check-AG scales linearly with respect to the number of time intervals for patterns S3 and S4. With the same number of observation time intervals, the higher the length of the trace, the longer it takes to complete the trace checking procedure.

5.2.3 Comparison with *SOLOIST-Translator*

Methodology. To answer RQ3, we compared the performance (in terms of execution time) of TemPsy-Check-AG with *SOLOIST-Translator*, the only publicly available tool for non-distributed trace checking of properties written in *SOLOIST*, a language that supports the same service provisioning patterns as *TemPsy-AG*.

However, the two non-distributed trace checking algorithms for *SOLOIST* proposed in the literature [BGKSP14, BBG+14], implemented in *SOLOIST-Translator*, do not scale well in terms of the length of the trace [BGK14]. A preliminary set of experiments that we conducted reported similar results to those published in [BGKSP14, BBG+14]: *SOLOIST-Translator* cannot handle traces longer than 1500 for checking properties with patterns S3 and S4, and traces longer than 1200 for checking properties with pattern S1. This is due to the internal translation of

input properties done by the tool, which takes into account the granularity of the timestamps of the trace elements. For these reasons, we could only use small traces to compare the two tools:

- For properties with pattern S1, we generated traces with length varying from 500 to 1100 in steps of 100.
- For properties with patterns S3 and S4, we generated traces with length varying from 1000 to 1500 in steps of 100, and varied the number of observation intervals (by setting the parameter h) through the values 2, 10, 50, 100.

Results. The results indicate that *SOLOIST-Translator* displays a steep linear growth of the execution time for traces with a short length, whereas our approach takes almost a constant time. For space reasons, we only show (in Figure 5d) the results for the case of properties with pattern S1 "average response time". For a trace with length 1100 (the largest length we considered), *SOLOIST-Translator* took 12864 ms, whereas TemPsy-Check-AG took 1243 ms. We remark that the execution time taken by *SOLOIST-Translator* for checking pattern S1 on a trace with length 1100 is *more than twice* the time (6250 ms) taken by TemPsy-Check-AG for checking a similar property on a 1M trace (see Figure 5a).

For the case of properties with the "average number of events" pattern, when varying the number of time intervals from 2 to 100, the execution time of *SOLOIST-Translator* ranges from 7929 ms to 8285 ms for a trace with length 1000, and from 17614 ms to 18 254 ms for a trace with length 1500. On the other hand, TemPsy-Check-AG takes from 1260 ms to 1446 ms for a trace with length 1000, and from 1247 ms to 1574 ms for a trace with length 1500. We observed similar values for the case of properties with the "maximum number of events".

The answer to RQ3 is that TemPsy-Check-AG can handle much larger traces than *SOLOIST-Translator* (with up to a 1000x increase in length) and exhibits faster execution times.

5.3 Discussion

The results presented above indicate that the execution time of TemPsy-Check-AG is acceptable from a practical standpoint: temporal properties with aggregation operators can be checked on a trace with millions of events within seconds. Furthermore, the comparison with *SOLOIST-Translator*, a state-of-the-art tool that supports the same service provisioning patterns (S1, S3, S4) as TemPsy-Check-AG, shows that our approach can handle much larger traces than *SOLOIST-Translator* (with up to a 1000x increase in length) and exhibits faster execution times. Overall, these results show that extending a model-driven trace checking approach [DBB17a] with support for a larger range of properties yield a scalable and viable solution for verifying, in offline settings, temporal properties with aggregation operators.

Threats to validity. One of the main threats is the use of synthesized traces in the evaluation. Real execution traces might be different, in terms of events occurrences and time distances. However, this threat does not affect our research questions 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. Another threat is the representativeness of the properties templates used for the evaluation. These property templates, although simple, represent the type of properties (with service provisioning patterns) that can be written in realistic scenarios, since they are based

on those extracted in a study [BGPS12] of specifications written in industrial settings; furthermore, similar templates have been used in existing work [BGKSP14, BBG$^+$14, BGS13] on trace checking of the same service provisioning patterns. 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. Finally, as for the comparison with *SOLOIST-Translator*, we remark that its specification language (*SOLOIST*) is more expressive than *TemPsy-AG* (e.g., by supporting first-order quantification and the full set of metric temporal logic modalities); hence the performance of *SOLOIST-Translator* could have been negatively affected by the more complex implementation needed to support a richer specification language.

6 Related work

Besides the work revolving around the *SOLOIST* language [BGKSP14, BBG$^+$14, BGS13], there are other approaches that focus on run-time verification of properties with aggregation operators [SW95, FSS05, DSS$^+$05, Rap16, BKMZ15]. Among the most recent, Basin et al. [BKMZ15] present an extension of the MFOTL logic that supports aggregation on data, i.e., terms used in the logical predicates, and the corresponding monitoring algorithm; Rapin [Rap16] proposes a dense-time specification language (and the monitoring algorithm), in which aggregation operators can be used to specify invariants of hybrid, signal-based systems. The main difference of these approaches from ours is the specific type of aggregation operators considered: in all the aforementioned approaches, the aggregation is done on the values of data/signals whereas the service provisioning patterns supported by *TemPsy-AG* aggregate events. LarvaStat [CGP10] is an extension of the Larva monitoring tool [CP17] with support for collecting statistical data of the execution, through point- and interval-statistics operators; however, the definition of these operators is quite operational, requiring explicitly to specify the update rules for the aggregations and the conditions characterizing the intervals. In contrast, *TemPsy-AG* provides high-level aggregation operators, with pre-defined semantics.

One of the key features of Complex Event Processing (CEP) systems [Luc01] is to aggregate data from multiple sources, using aggregating operators similar to those included in *TemPsy-AG*. One of the main difference between CEP approaches and RV ones is that the former compute the result of a query on a event trace, whereas the latter evaluate a property on a trace [Hal16]. Approaches like BeepBeep [Hal16] combine CEP and RV, allowing the evaluation of properties (possibly temporal) over event streams processed (e.g., by means of aggregating operators) through a CEP-like pipeline. Conceptually, *TemPsy-AG* adopts a similar approach, since in the OCL constraints the events are aggregated according to the pattern semantics before the evaluation of the relational expression in the property.

7 Conclusion and Future Work

The verification of complex software systems often requires to check temporal properties that contain aggregation operators. When specifying such properties, a software engineer can leverage an existing catalogue of property specification patterns, called “service provisioning patterns” [BGPS12]. Nevertheless, existing solutions for trace checking of these properties suffer from scalability limitations. In this paper we have

presented our solution for scalable trace checking of temporal properties with aggregation operators, extending an existing model-driven approach for trace checking. Our approach is based on an optimized mapping into OCL constraints (on a meta-model of execution traces) of the main service provisioning patterns. We have implemented our approach in the tool TemPsy-Check-AG and evaluated its performance: the results show that our approach can check temporal properties with aggregation operators on a trace with millions of event within seconds, scales linearly with respect to the length of the input trace, and can deal with much larger traces than a state-of-the-art tool. Furthermore, the results indicate the feasibility and viability of extending model-driven trace checking [DBB17a], a promising run-time verification approach enabled by MDE technologies, with support for a larger class of properties, while retaining acceptable performance from a practical standpoint.

As part of future work, we plan to collaborate with an industrial partner, to apply (and extend) *TemPsy-AG* for the verification of properties of cyber physical systems. We will also investigate how to leverage Big Data technologies to parallelize our trace checking approach, inspired by the existing work on distributed trace checking [BBG^{+}16, BGK14, BCE^{+}14, BKSB^{+}12].

References

[BBG^{+}14] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. SMT-based checking of SOLOIST over sparse traces. In *FASE2014*, volume 8411 of *LNCS*, pages 276–290. Springer, 2014.

[BBG^{+}16] Marcello M. Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Efficient large-scale trace checking using mapreduce. In *Proc. ICSE2016*, pages 888–898. ACM, 2016.

[BCE^{+}14] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. In *Proc. RV2014*, volume 8734 of *LNCS*, pages 31–47. Springer, 2014.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition.* Morgan & Claypool Publishers, 2nd edition, 2017.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.

[BGK14] Domenico Bianculli, Carlo Ghezzi, and Srđan Krstić. Trace checking of metric temporal logic with aggregating modalities using MapReduce. In *Proc. SEFM2014*, volume 8702 of *LNCS*, pages 144–158. Springer, 2014.

[BGKSP14] Domenico Bianculli, Carlo Ghezzi, Srdan Krstic, and Pierluigi San Pietro. Offline trace checking of quantitative properties of service-based applications. In *Proc. SOCA2014*, pages 9–16. IEEE, 2014.

[BGPS12] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study

in service-based applications. In *Proc. ICSE2012*, pages 968–976. IEEE, 2012.

[BGS13] Domenico Bianculli, Carlo Ghezzi, and Pierluigi San Pietro. The tale of SOLOIST: a specification language for service compositions interactions. In *Proc. FACS2012*, volume 7684 of *LNCS*, pages 55–72. Springer, 2013.

[BKMZ15] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal methods in system design*, 46(3):262–285, 2015.

[BKSB+12] Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé. MapReduce for parallel trace validation of LTL properties. In *Proc. RV2012*, volume 7687 of *LNCS*, pages 184–198. Springer, 2012.

[CGP10] Christian Colombo, Andrew Gauci, and Gordon Pace. Larvastat: Monitoring of statistical properties. In *Proc. RV2010*, volume 6418 of *LNCS*, pages 480–484. Springer, 2010.

[CP17] Christian Colombo and Gordon Pace. Runtime verification using larva. In *Proc. RV-CuBES2017*, volume 3 of *Kalpa Publications in Computing*, pages 55–63. EasyChair, 2017.

[CZss] C. Czepa and U. Zdun. On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language. *IEEE Transactions on Software Engineering*, in press. doi: 10.1109/TSE.2018.2859926.

[DAC99] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proc. ICSE1999*, pages 411–420. ACM, 1999.

[DBB17a] Wei Dou, Domenico Bianculli, and Lionel Briand. A model-driven approach to trace checking of pattern-based temporal properties. In *Proc. MODELS2017*, pages 323–333. IEEE Computer Society, 2017.

[DBB17b] Wei Dou, Domenico Bianculli, and Lionel Briand. TemPsy-Check: a tool for model-driven trace checking of pattern-based temporal properties. In *Proc. RV-CuBES2017*, volume 3 of *Kalpa Publications in Computing*, pages 64–70. EasyChair, September 2017.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[Dou16] Wei Dou. *A Model-Driven Approach to Offline Trace Checking of Temporal Properties*. PhD thesis, University of Luxembourg, 2016. URL: http://hdl.handle.net/10993/29184.

[DSS+05] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. Lola: runtime monitoring of synchronous systems. In *Proc. TIME 2005*, pages 166–174. IEEE, june 2005.

[FHR13] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, volume 34

of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

[FSS05] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27:253–274, 2005.

[GL06] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electron. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.

[Hal16] Sylvain Hallé. When RV meets CEP. In *Proc. RV2016*, volume 10012 of *LNCS*, pages 68–91. Springer, 2016.

[KC05] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proc. ICSE2005*, pages 372–381. ACM, 2005.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, May/June 2009.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Rap16] Nicolas Rapin. Reactive property monitoring of hybrid systems with aggregation. In *Proc. RV2016*, volume 10012 of *LNCS*, pages 447–453. Springer, 2016.

[SW95] A Prasad Sistla and Ouri Wolfson. Temporal conditions and integrity constraints in active database systems. In *ACM SIGMOD Record*, volume 24, pages 269–280. ACM, 1995.

About the authors

Chaima Boufaied is a PhD student at the SnT Centre of the University of Luxembourg. Contact her at chaima.boufaied@uni.lu.

Domenico Bianculli is a research scientist at the SnT Centre of the University of Luxembourg. Contact him at domenico.bianculli@uni.lu.

Lionel Briand is a professor at the SnT Centre of the University of Luxembourg. Contact him at lionel.briand@uni.lu.

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) and from the University of Luxembourg (grant "MOVIDA").