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

Chaima BOUFAIED, Domenico BIANCULLI, Lionel BRIAND

ECMFA'19

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

Chaima BOUFAIED, Domenico BIANCULLI, Lionel BRIAND

ECMFA'19

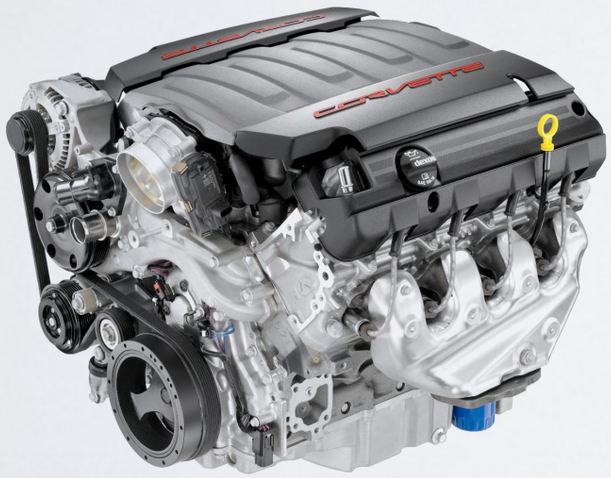
Runtime Verification

- **Automated specification-based technique:**
  - **A monitor evaluates the correctness of temporal properties of a particular execution**
- **(Offline) trace checking is a sub-type of run-time verification**

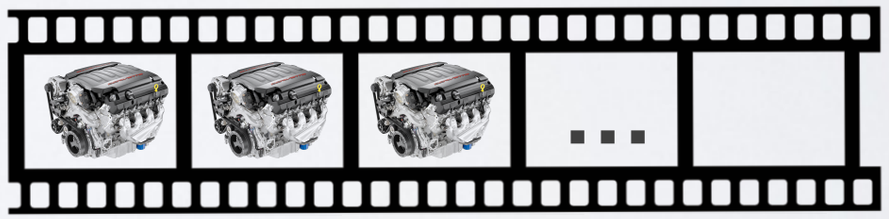
Execution Traces

Logging framework

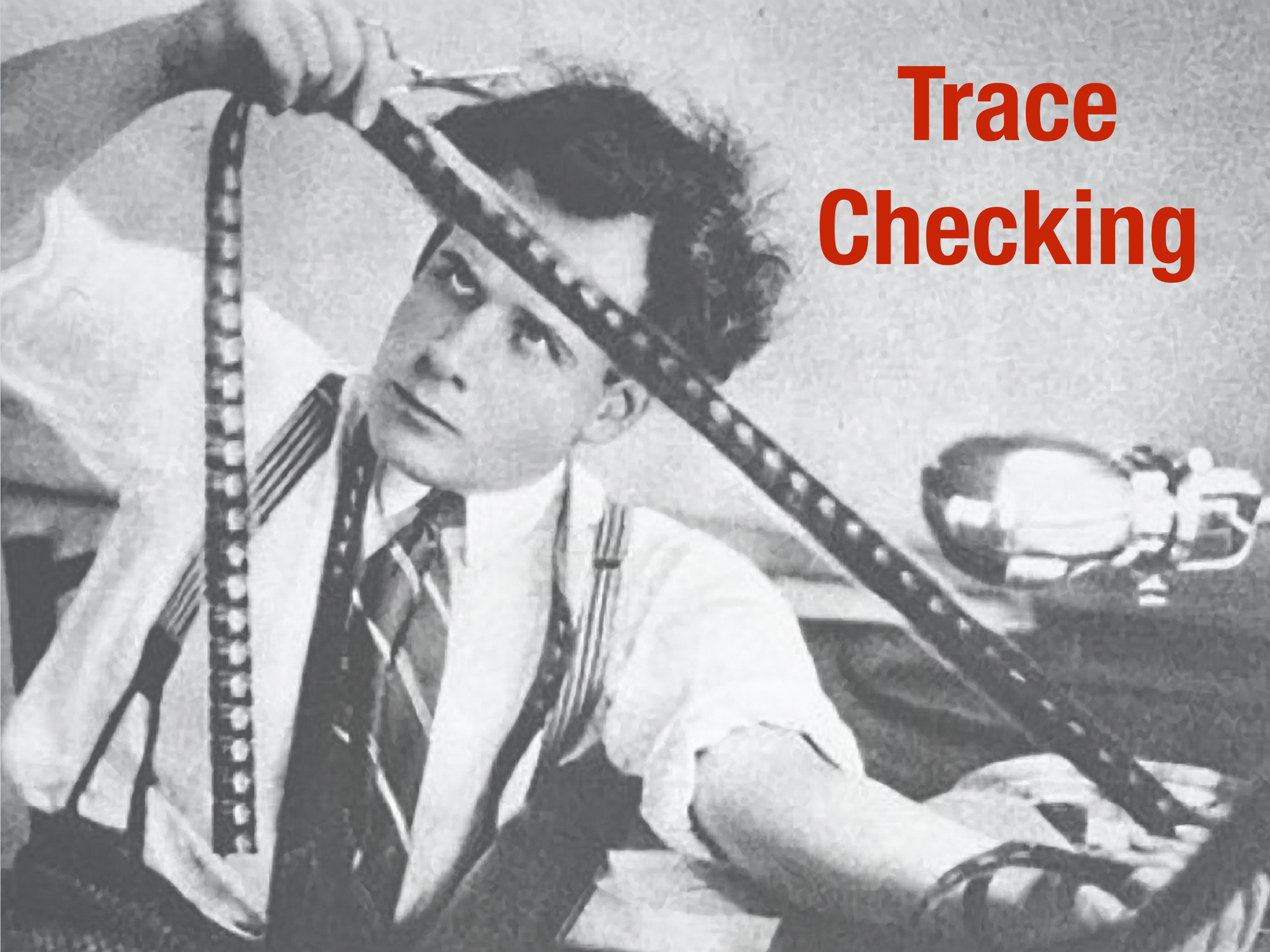
System



Execution trace



Trace Checking



Specification of Temporal Properties

Temporal Logic

```
(!P W (P W (!P W (P W []!P))))

<>R -> ((!P & !R) U (R | ((P & !R) U
          (R | ((!P & !R) U (R | ((P & !R) U
            (R | (!P U R)))))))))

<>Q -> (!Q U (Q & (!P W (P W (!P W (P W []!P))))))

[]((Q & <>R) ->
   ((!P & !R) U (R | ((P & !R) U
     (R | ((!P & !R) U (R | ((P & !R) U
       (R | (!P U R)))))))))

[](Q -> ((!P & !R) U (R | ((P & !R) U
          (R | ((!P & !R) U (R | ((P & !R) U
            (R | (!P W R) | []P)))))))))
```

Domain-specific Language (DSL)

globally at most 2 P
before R at most 2 P
after Q at most 2 P
between Q and R at most 2 P
after Q until R at most 2 P

- Syntax close to the natural language
- Does not require a strong mathematical background
- Important especially for adoption among practitioners

Specification Language: TemPsy Temporal Properties made easy

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

MODELS 2017

Temporal Properties in TemPsy

- **A TemPsy property is essentially based on:**

Scope	Pattern
Indicates the portion(s) of a system execution to consider (e.g., before, between-and)	A high-level abstraction of a behavior (e.g., universality, response)

Temporal Properties in TemPsy

- **A TemPsy property is essentially based on:**

Scope

Indicates the portion(s) of a system execution to consider
(e.g., before, between-and)

Pattern

A high-level abstraction of a behavior
(e.g., universality, response)

an extension of Dwyer et al.'s system of property specification patterns

TemPsy example - Request

“Once a card request is approved, the applicant is notified within three days; this notification has to occur before the production of the card is started.”

(Before + Response)

TemPsy example - Request

“Once a card request is approved, the applicant is notified within three days; this notification has to occur before the production of the card is started.”

(Before + Response)

TemPsy example - Request

“Once a card request is approved, the applicant is notified within three days; this notification has to occur before the production of the card is started.”

(Before + Response)

TemPsy example - Request

“Once a card request is approved, the applicant is notified within three days; this notification has to occur before the production of the card is started.”

(Before + Response)

TemPsy example - Request

```
temporal R1:

 before ICM.issueCard

 ICM.notifyApproval responding at most 3*24*3600 tu
 ICM.approveRequest;
```

“Once a card request is approved, the applicant is notified within three days; this notification has to occur before the production of the card is started”

Model-driven Trace Checking with TemPsy

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

MODELS 2017

Why model-driven Trace Checking?

- **Industrial contexts where model-driven engineering is the practice**

- **any software solution added to the development process (e.g., trace checking) should adhere to OMG specifications and rely on the corresponding tools**

Model-driven Trace Checking

Problem of checking a temporal property on a trace

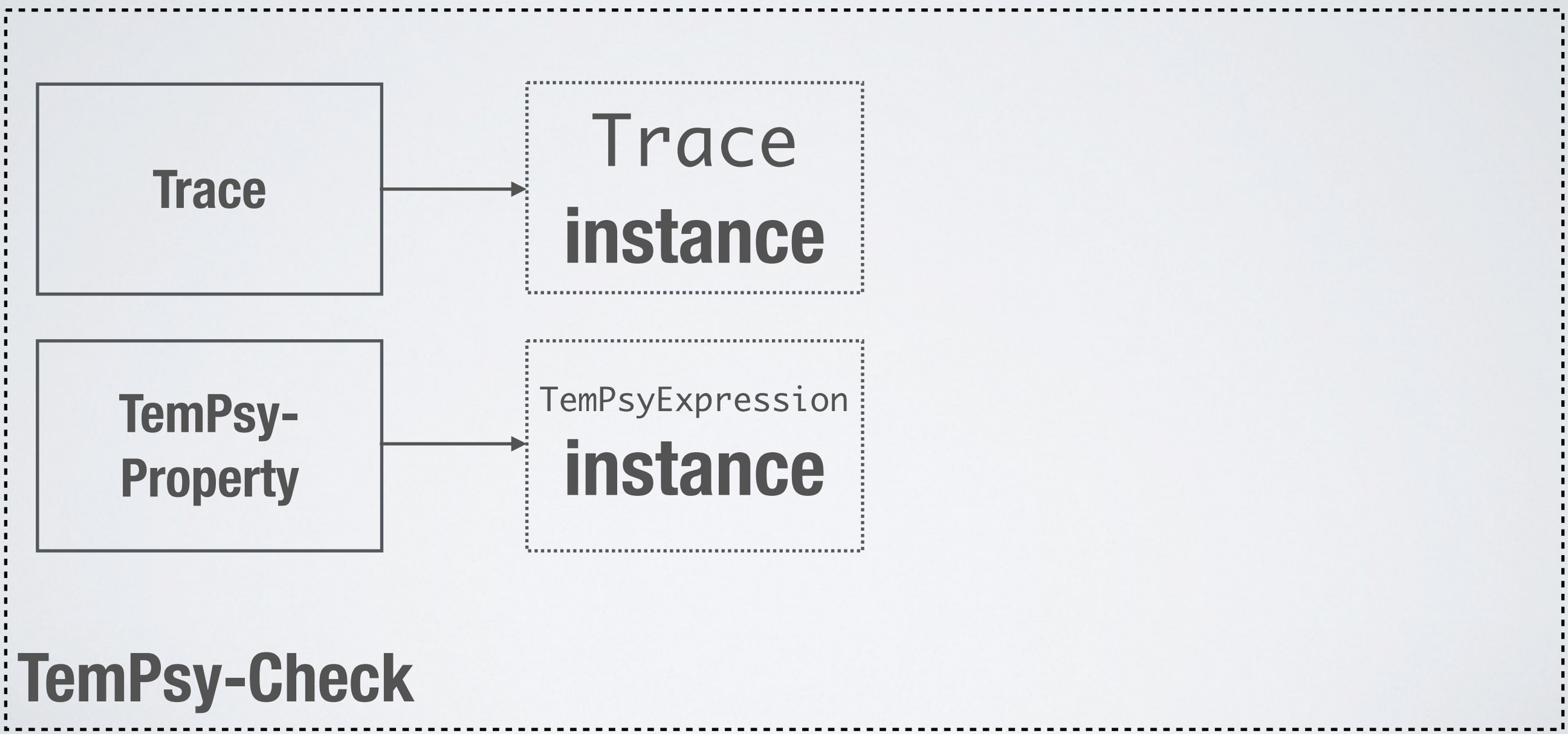
Equivalent to the temporal property

Reduction

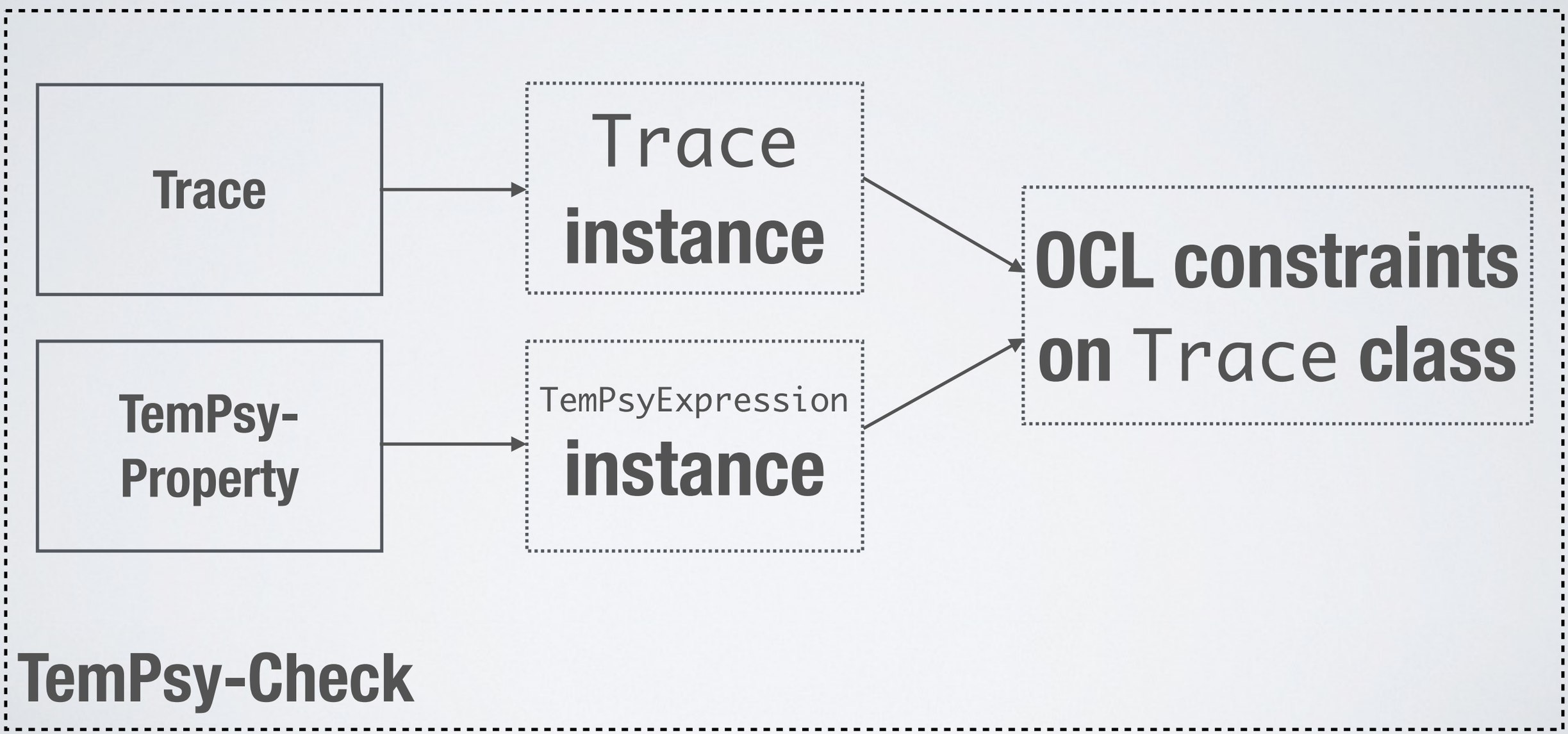
Evaluation of an OCL constraint on an instance of the trace meta-model

Model-driven Offline Trace Checking of TemPsy Properties

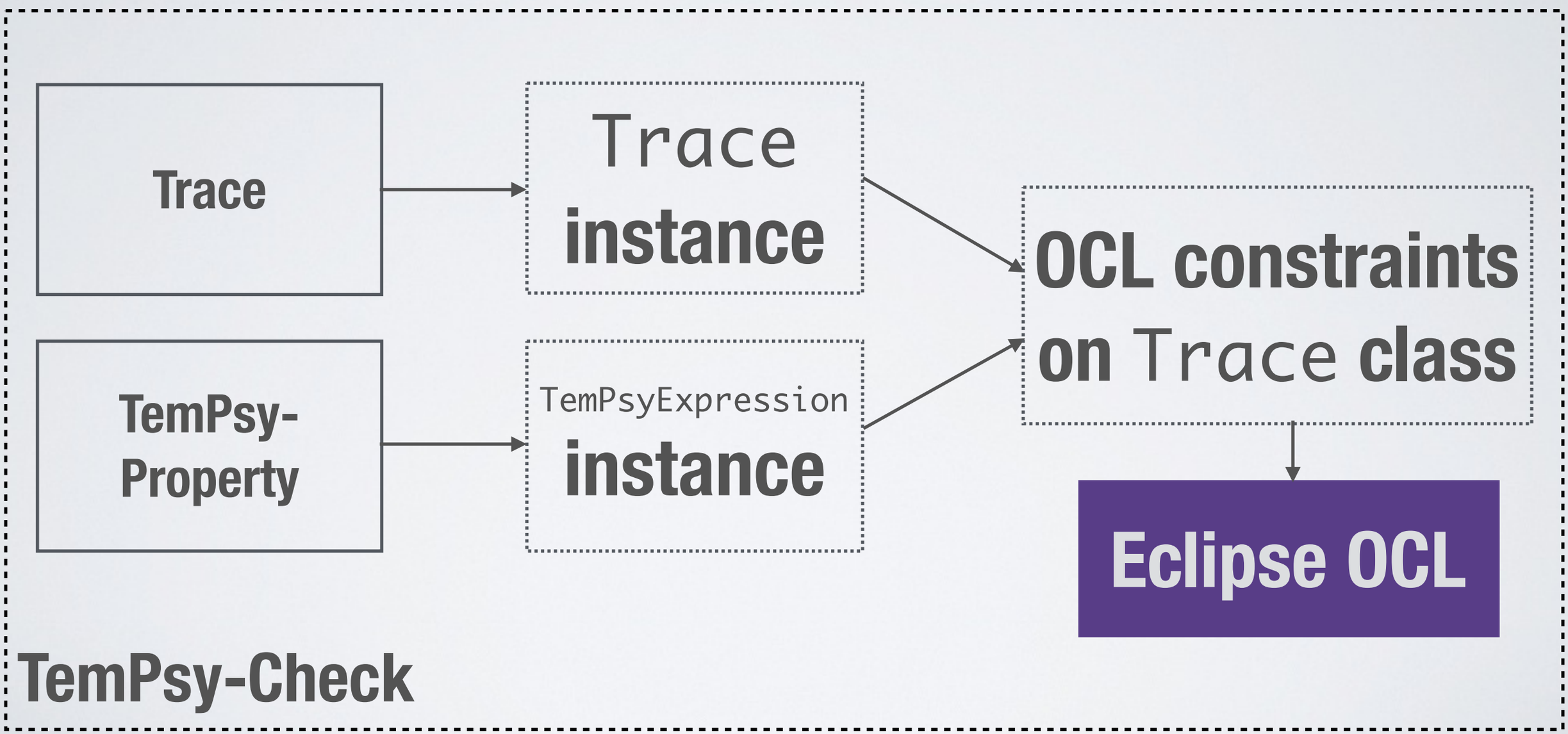
Model-driven Offline Trace Checking of TemPsy Properties



Model-driven Offline Trace Checking of TemPsy Properties



Model-driven Offline Trace Checking of TemPsy Properties



TemPsy Expressiveness Limitation

- It only supports patterns from Dwyer et al.’s system
  - There are other specification patterns systems out there…

Service Provisioning Patterns

- Classification of 900+ requirements in the context of service-based applications
- Almost 82% of industrial specifications analyzed in the study represent temporal properties with aggregation operators

Specification Patterns from Research to Industry: A Case Study in Service-Based Applications

Domenico Bianculli
Faculty of Informatics
University of Lugano
Lugano, Switzerland
domenico.bianculli@usi.ch

Carlo Ghezzi
DEEPSE group - DEI
Politecnico di Milano
Milano, Italy
ghezzi@elet.polimi.it

Cesare Pautasso
Faculty of Informatics
University of Lugano
Lugano, Switzerland
cesare.pautasso@usi.ch

Patrick Senti
Information Technology
Credit Suisse AG
Zürich, Switzerland
patrick.senti@credit-suisse.com

***Abstract*—Specification patterns have proven to help developers to state precise system requirements, as well as formalize them by means of dedicated specification languages. Most of the past work has focused its applicability area to the specification of concurrent and real-time systems, and has been limited to a research setting. In this paper we present the results of our study on specification patterns for service-based applications (SBAs). The study focuses on industrial SBAs in the banking domain. We started by performing an extensive analysis of the usage of specification patterns in published research case studies — representing almost ten years of research in the area of specification, verification, and validation of SBAs. We then compared these patterns with a large body of specifications written by our industrial partner over a similar time period. The paper discusses the outcome of this comparison, indicating that some needs of the industry, especially in the area of requirements specification languages, are not fully met by current software engineering research.**

***Keywords*-specification patterns; specification languages; requirements specifications; services**

I. Introduction

The concept of *pattern* has been initially proposed in the domain of architecture by C. Alexander [1], to represent "the description of a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

This idea of pattern has then been adopted in software engineering with the concept of *design patterns* [2], as reusable solutions for recurring problems in software design. Subsequently, the concept of design patterns has been embraced in different sub-domains of software engineering, from architectural patterns to reengineering patterns, including property specification patterns.

Property specification patterns [3] have been proposed in the late '90s in the context of finite-state verification, as a means to express recurring properties in a generalized form, which could be formalized in different specification languages, such as temporal logic. Specification patterns aimed at bridging the gap between finite-state verification tools (e.g., model checkers) and practitioners, by providing the latter with a powerful instrument for writing down properties to be fed to a formal verification tool.

Given the origin of property specification patterns, most of past work has focused its applicability area to the specification (and the verification) of concurrent and real-time systems (see, for example, [4]), with limited applications outside the research setting.

In the last years, open software [5] systems such as service-based applications (SBAs) have emerged, introducing new engineering challenges due to their dynamic and decentralized nature. One of these research challenges is related to the specification, verification and validation of SBAs [6]. At the same time, service-oriented architectures (SOAs) have gained a lot of attention in enterprises, which started to adopt them for the integration of their information systems [7]. However, to the best of our knowledge, the research literature on specification, verification and validation of SBAs has presented limited evidence of its applicability to and suitability for industrial-level case studies.

One of the questions that we asked ourselves during our research is whether existing requirements specification languages are expressive enough to formalize common requirements specifications used in industry. In particular, we are interested in evaluating the use of specification patterns for expressing properties of industrial SBAs, to assess if existing and well-known specification patterns are adequate or not. If this is not the case, our goal is to gather substantial evidence for new specification patterns and/or language constructs required to support their practical use in industrial settings.

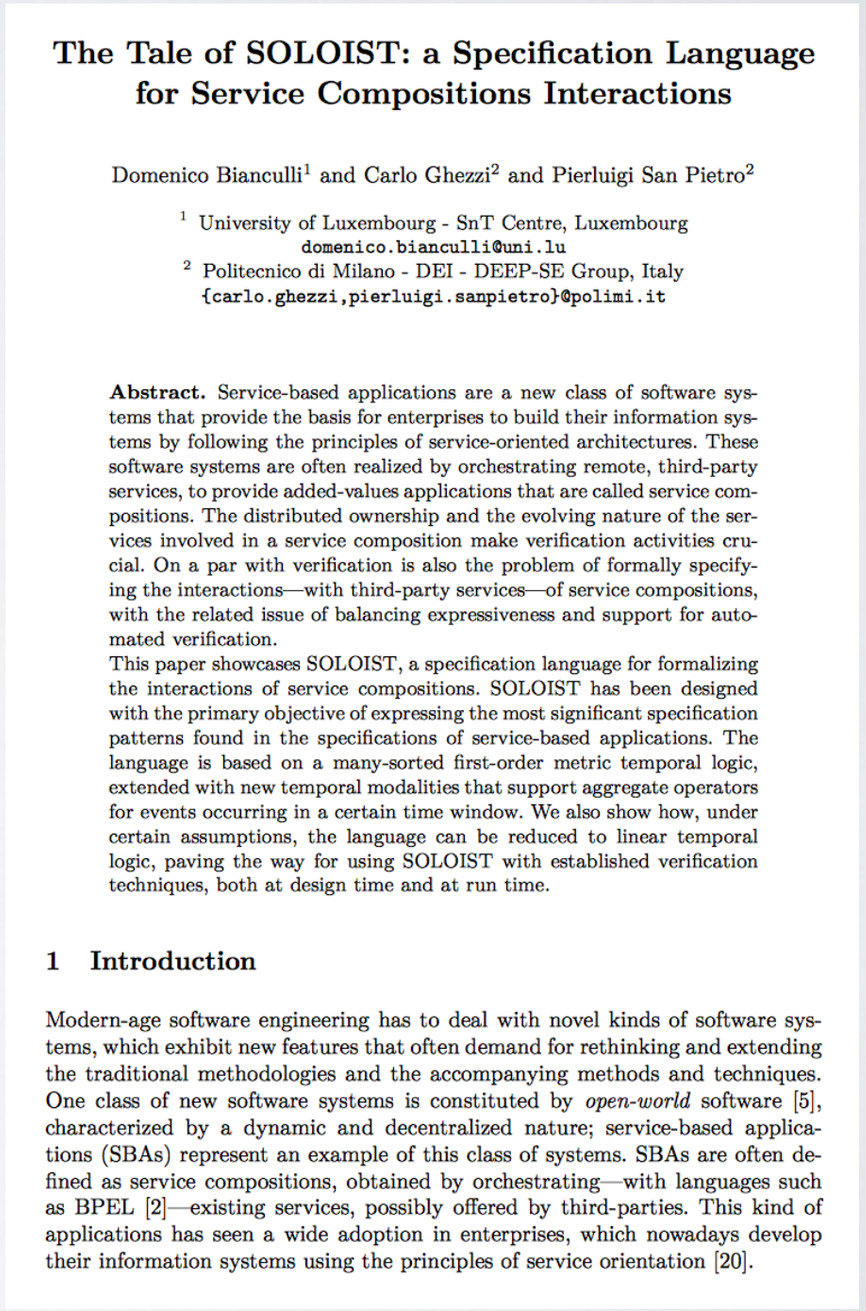
In this paper we present the results of our study on the use of specification patterns in SBAs. The study has been performed by analyzing the requirements specifications of two sets of case studies. One set was composed of case studies extracted from research papers in the area of specification, verification and validation of SBAs, which appeared in the main publishing venues of software engineering and service-oriented computing within the last 10 years. The other set was composed of case studies corresponding to

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012

SOLOIST

- A specification language for formalizing the interactions of service compositions
- It supports the service provisioning patterns
  - Average response time
  - Counting the number of events
  - Average number of events
  - Maximum number of events
  - Absolute time
  - Elapsed time
  - Data awareness



The Tale of SOLOIST: a Specification Language for Service Compositions Interactions

Domenico Bianculli[1] and Carlo Ghezzi[2] and Pierluigi San Pietro[2]

[1] University of Luxembourg - SnT Centre, Luxembourg
domenico.bianculli@uni.lu
[2] Politecnico di Milano - DEI - DEEP-SE Group, Italy
{carlo.ghezzi,pierluigi.sanpietro}@polimi.it

Abstract. Service-based applications are a new class of software systems that provide the basis for enterprises to build their information systems by following the principles of service-oriented architectures. These software systems are often realized by orchestrating remote, third-party services, to provide added-values applications that are called service compositions. The distributed ownership and the evolving nature of the services involved in a service composition make verification activities crucial. On a par with verification is also the problem of formally specifying the interactions—with third-party services—of service compositions, with the related issue of balancing expressiveness and support for automated verification.
This paper showcases SOLOIST, a specification language for formalizing the interactions of service compositions. SOLOIST has been designed with the primary objective of expressing the most significant specification patterns found in the specifications of service-based applications. The language is based on a many-sorted first-order metric temporal logic, extended with new temporal modalities that support aggregate operators for events occurring in a certain time window. We also show how, under certain assumptions, the language can be reduced to linear temporal logic, paving the way for using SOLOIST with established verification techniques, both at design time and at run time.

1 Introduction

Modern-age software engineering has to deal with novel kinds of software systems, which exhibit new features that often demand for rethinking and extending the traditional methodologies and the accompanying methods and techniques. One class of new software systems is constituted by *open-world* software [5], characterized by a dynamic and decentralized nature; service-based applications (SBAs) represent an example of this class of systems. SBAs are often defined as service compositions, obtained by orchestrating—with languages such as BPEL [2]—existing services, possibly offered by third-parties. This kind of applications has seen a wide adoption in enterprises, which nowadays develop their information systems using the principles of service orientation [20].

Motivations

- Requirements specifications based on temporal properties with aggregation operators are common
- The support in terms of verification of such properties is limited
  - The only language that supports such temporal properties is SOLOIST
    - The corresponding tool **does not scale** with respect to the **length of the trace!**

Goal of this work

- **Bridging the gap between property specifications based on service provisioning patterns and model-driven trace checking**

- **Expressiveness: extension of TemPsy language**

- **Checking: extension of TemPsy-Check procedure**

TemPsy-AG:
Our TemPsy Extension

SOLOIST Patterns supported by TemPsy-AG

- Maximum number of events
- Average number of events
- Average response time

SOLOIST Patterns supported by TemPsy-AG

- Maximum number of events

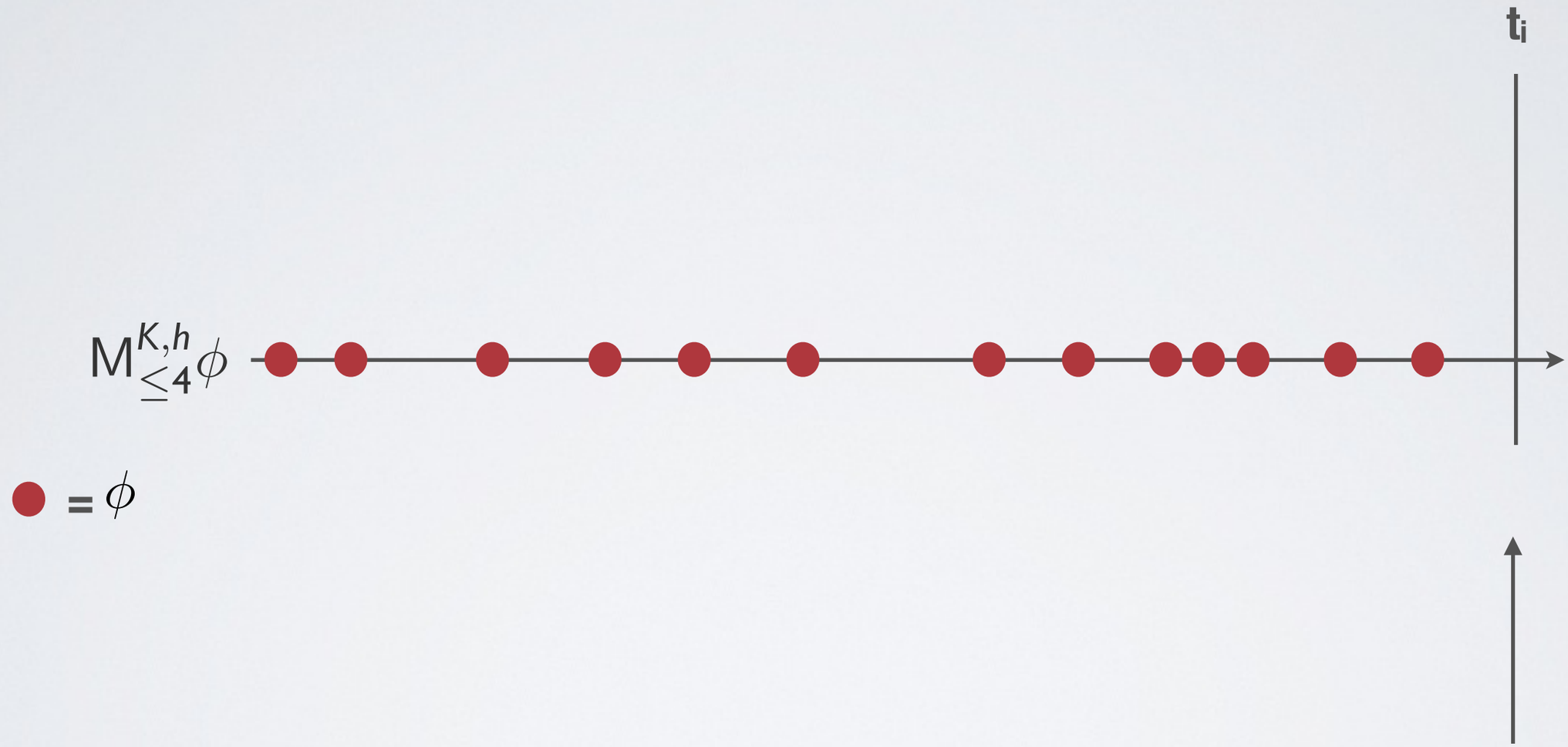
The maximum number of client requests per hour computed over the daily business hours should be less than 10000

- Average number of events
- Average response time

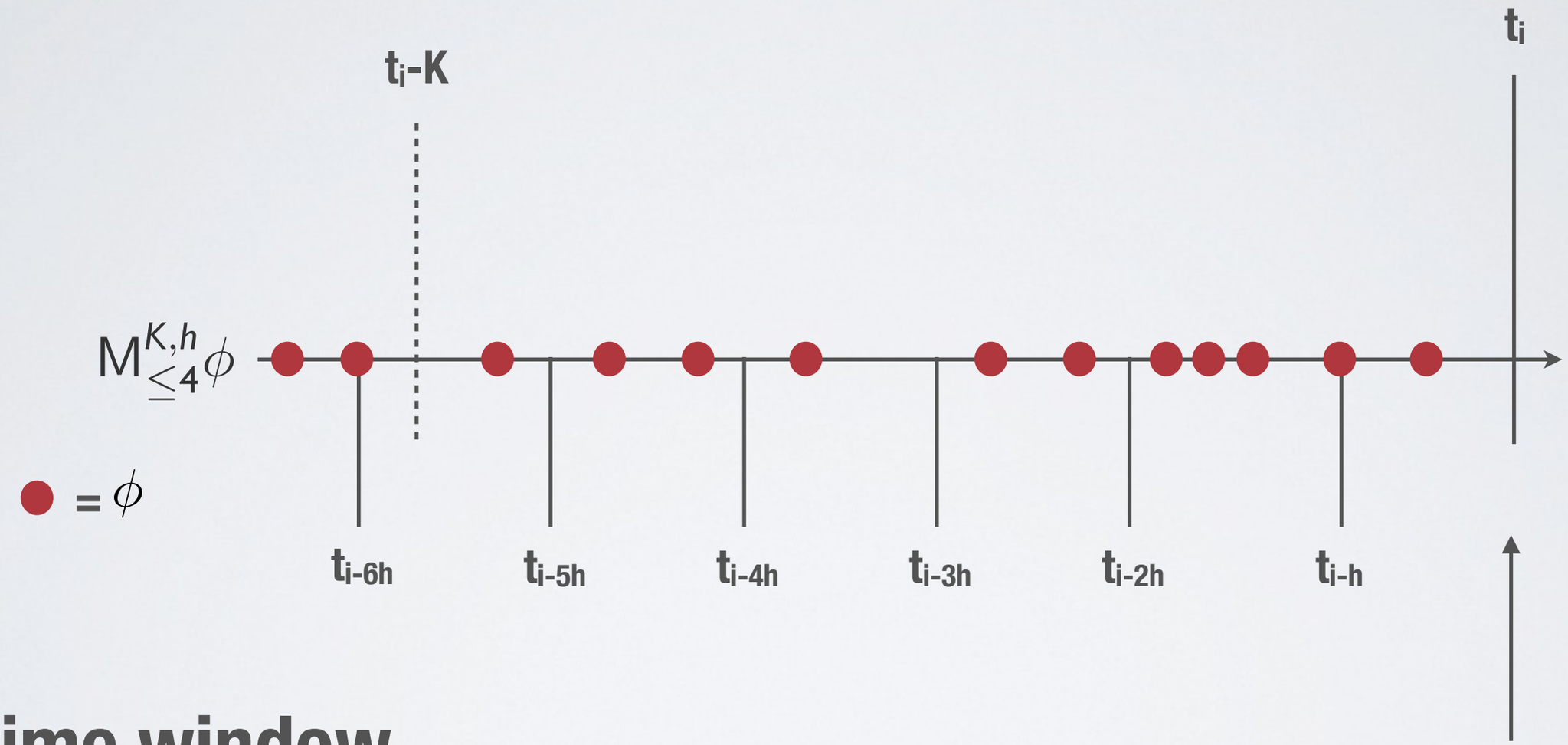
“Maximum number of events” pattern

● = ϕ

“Maximum number of events” pattern



“Maximum number of events” pattern

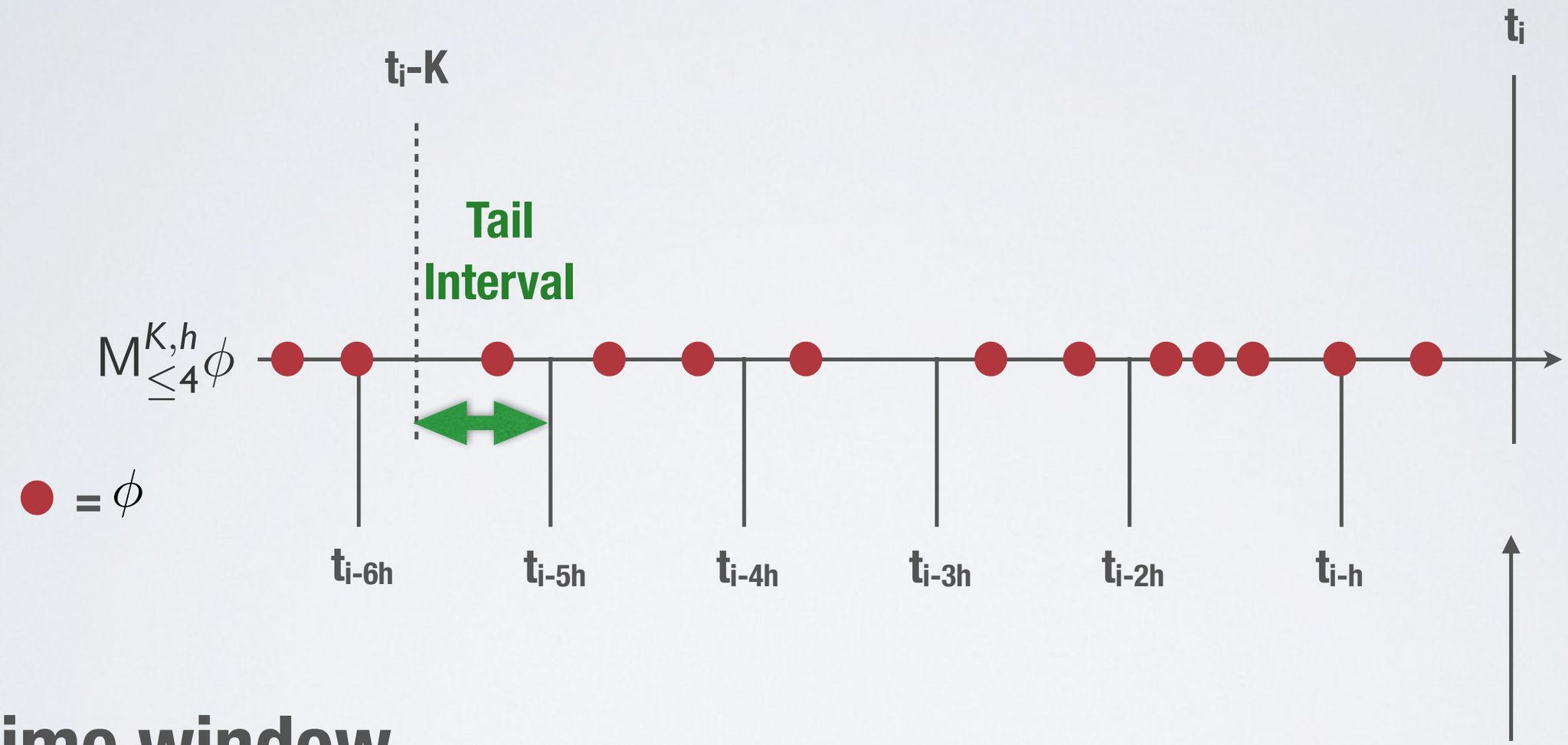


K= time window

h= time interval

Time instant at which the property is evaluated

“Maximum number of events” pattern



K= time window

h= time interval

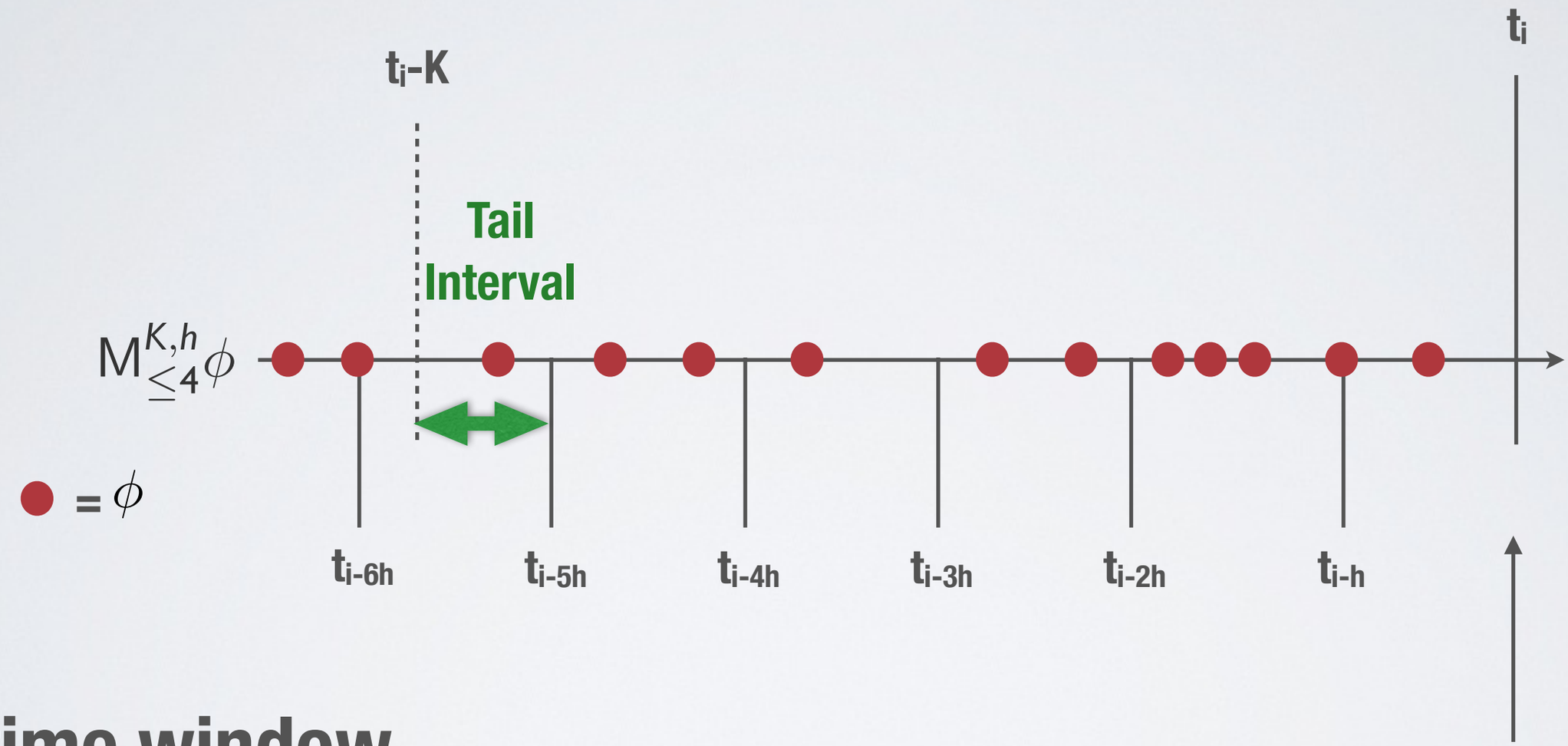
Time instant at which the property is evaluated

SOLOIST Patterns supported by TemPsy-AG

- Maximum number of events
- Average number of events
- Average response time

The average number of client requests per hour computed over the daily business hours should be less than 10000

“Maximum number of events” pattern

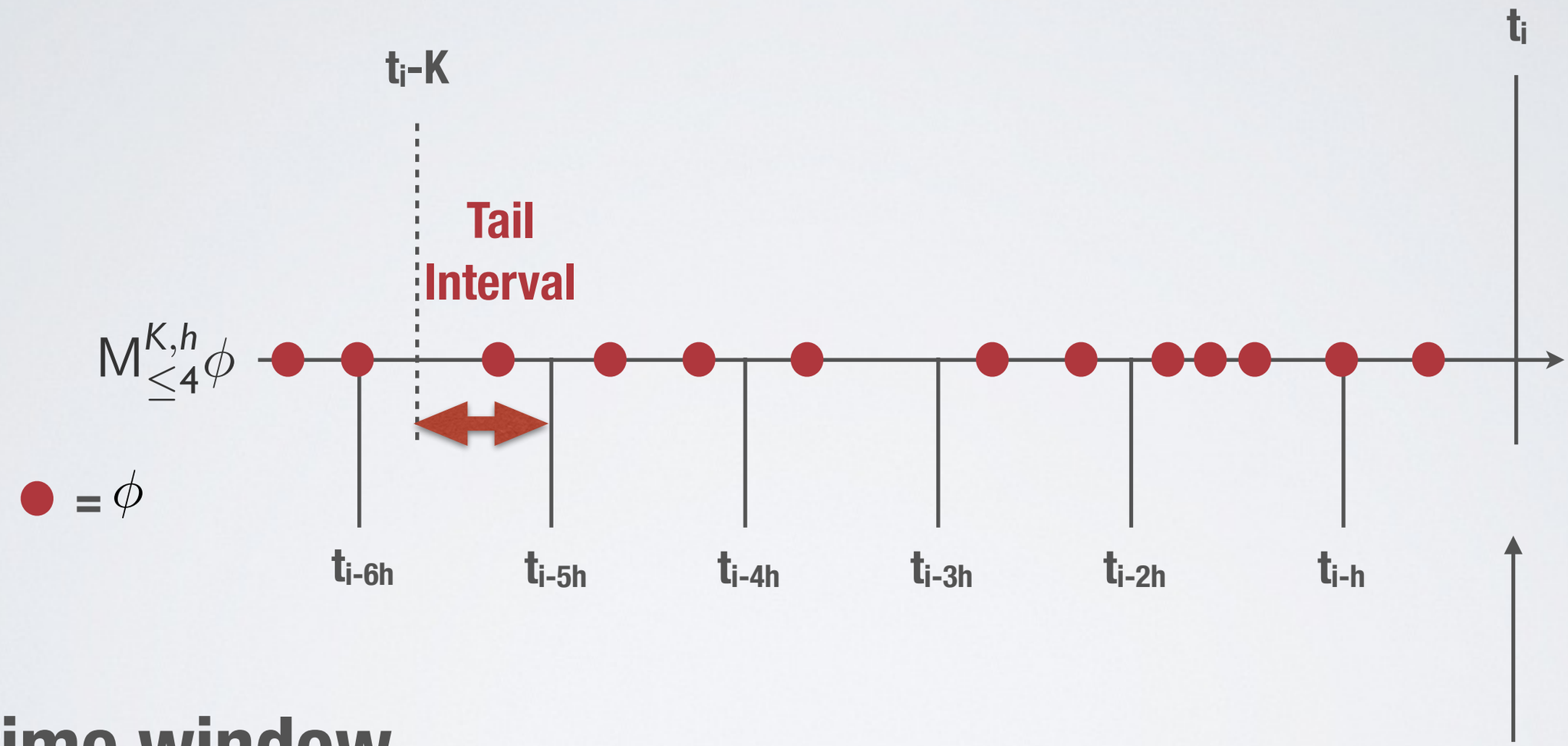


K= time window

h= time interval

Time instant at which the property is evaluated

“Average number of events” pattern



K= time window

h= time interval

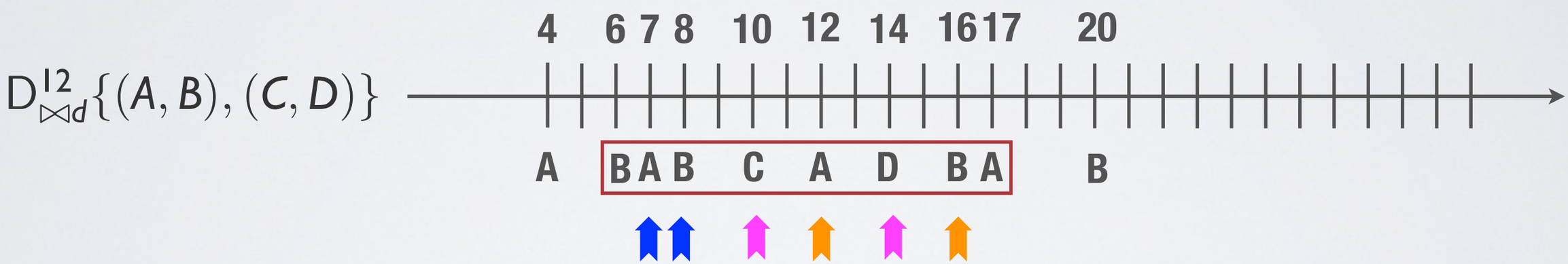
Time instant at which the property is evaluated

SOLOIST Patterns supported by TemPsy-AG

- Maximum number of events
- Average number of events
- Average response time

The average distance between event “client request” and “client confirmation” in the last 2 hours should be less than 3

“Average response time” pattern



$$\frac{(8-7)+(16-12)+(14-10)}{3} \bowtie d$$

Excerpt from TemPsy-AG Grammar for the supported SOLOIST Patterns

<Pattern> ::= ‘maximum’ <Event> ‘within’ <TimeUnitExpression>
‘every’ <TimeUnitExpression> <RO> <Int>

| ‘average’ <Event> ‘within’ <TimeUnitExpression>
‘every’ <TimeUnitExpression> <RO> <Int>

| ‘avgET for pair(s)’ ‘(‘ <event> ‘,’ <event> ’)’+ ‘within’
<TimeUnitExpression> <RO> <Int>

<TimeUnitExpression> ::= <Int> ‘tu’
<RO> ::= ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘==’ | ‘<>’

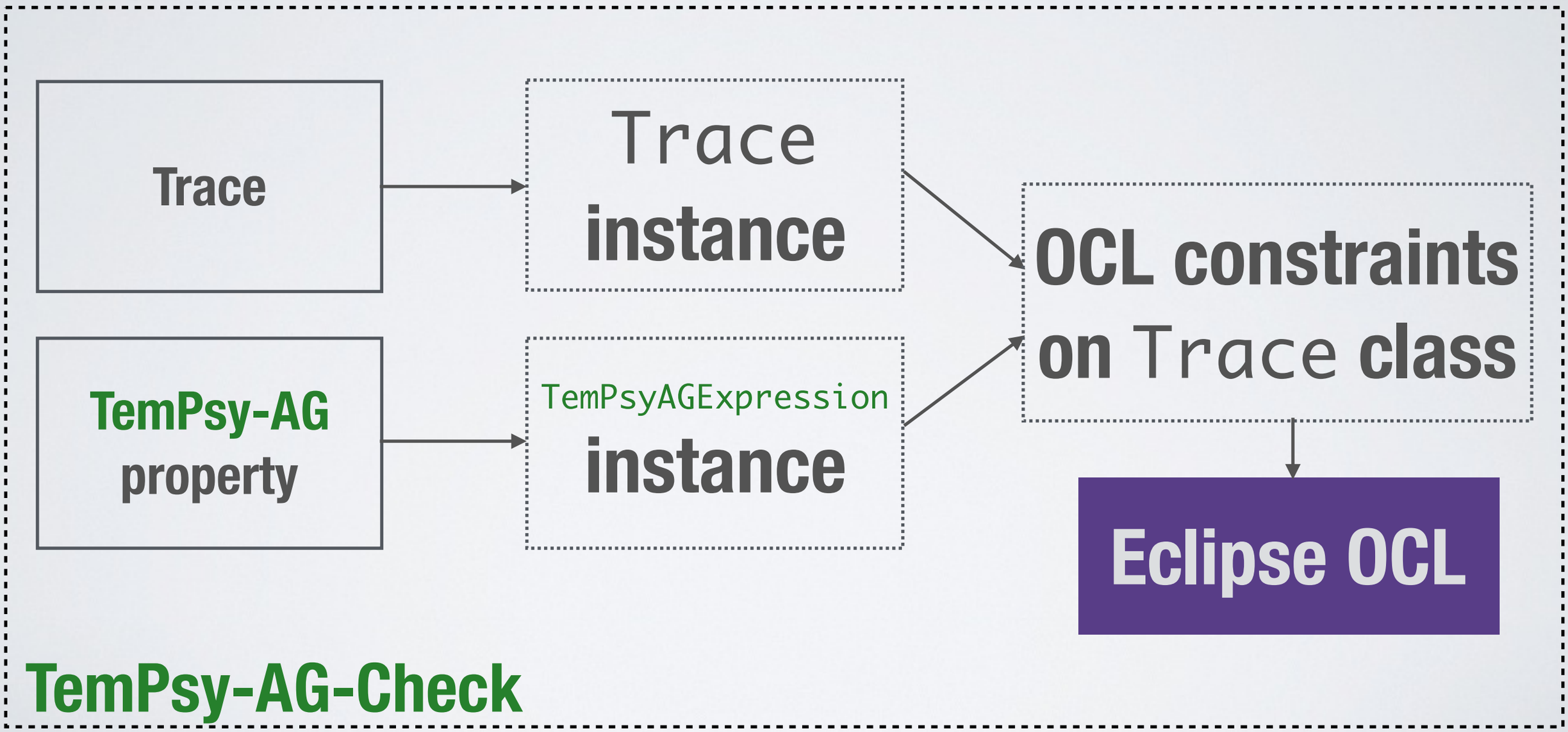
Examples of Properties in TemPsy-AG

```
P1:

 globally maximum actA within 720 tu every 60 tu < 5;
```

“The maximum number of invocations, in a time interval of 60 time units, of activity actA during the past 720 time units should be less than 5”

Trace Checking Extension

Model-driven Offline Trace Checking of TemPsy-AG Properties



Simplified Template of OCL Constraints

```
context Trace
inv: let subtraces=applyScope*S*(scope) in
        subtraces->forAll(subtrace |
        checkPattern*P*(subtrace, pattern))
```

Simplified Template of OCL Constraints

```
context Trace
inv: let subtraces=applyScope*S*(scope) in
        subtraces->forAll(subtrace |
        checkPattern*P*(subtrace, pattern))
```

Simplified Template of OCL Constraints

```
context Trace
inv: let subtraces=applyScope*S*(scope) in
        subtraces->forAll(subtrace |
        checkPattern*P*(subtrace, pattern))
```

Extensions of OCL Constraints

- checkPatternMaximum
- checkPatternAverage
- checkPatternAverageElapsedTime

checkPatternMaximum

Iteration

```
def: checkPatternMaximum(subtrace:OrderedSet(trace::TraceElement), pattern:tempsy::Pattern):Boolean =
-- check the satisfiability of the maximum pattern
fromTimeStamp:Integer = subtrace->last().timestamp,
if v.finalMax = expNbr and (tempsy::RelationalOperator::EQL = operator ) or
endif
```

Dividing the time window into time intervals (including the tail interval)

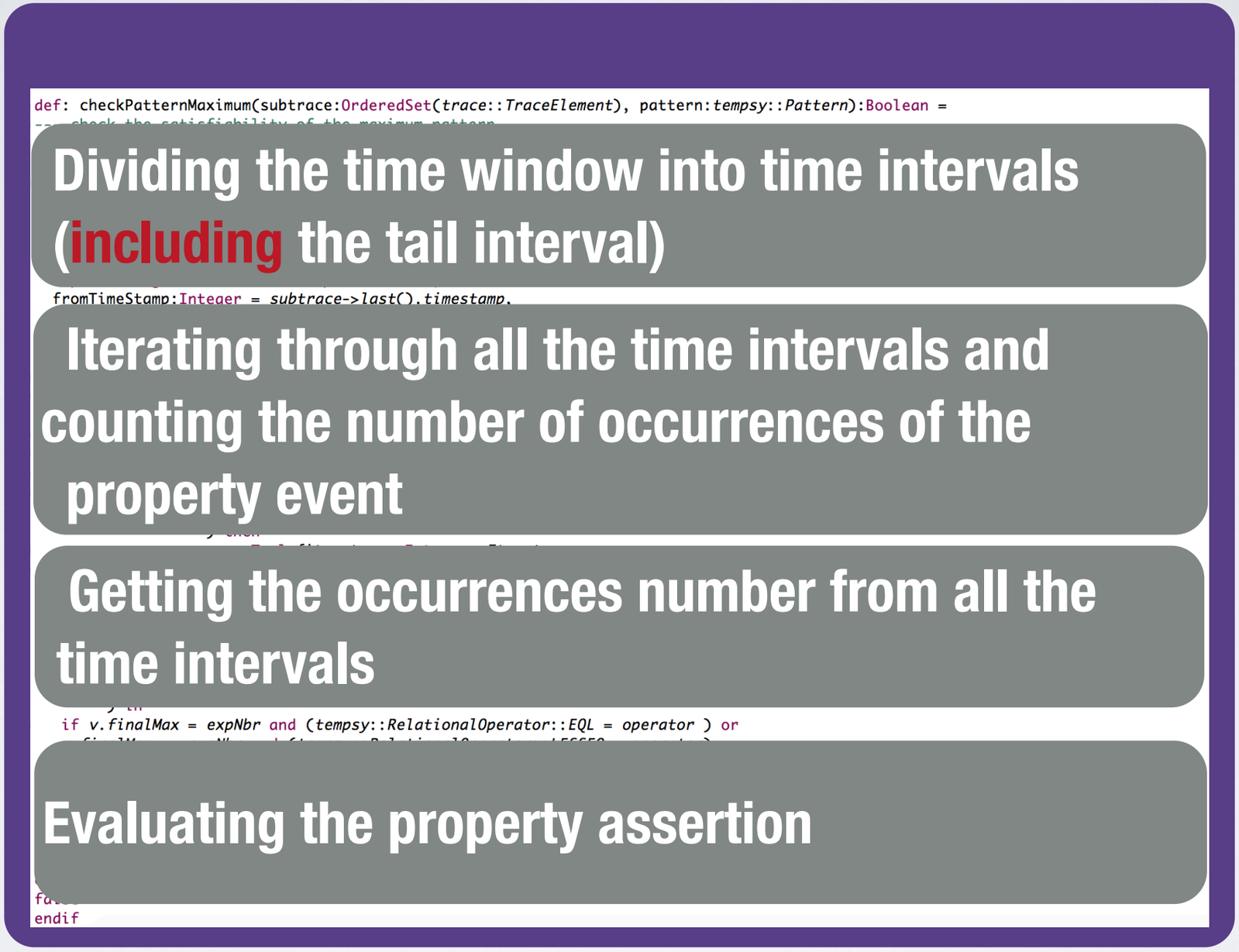
Iterating through all the time intervals and counting the number of occurrences of the property event

Getting the occurrences number from all the time intervals

Evaluating the property assertion

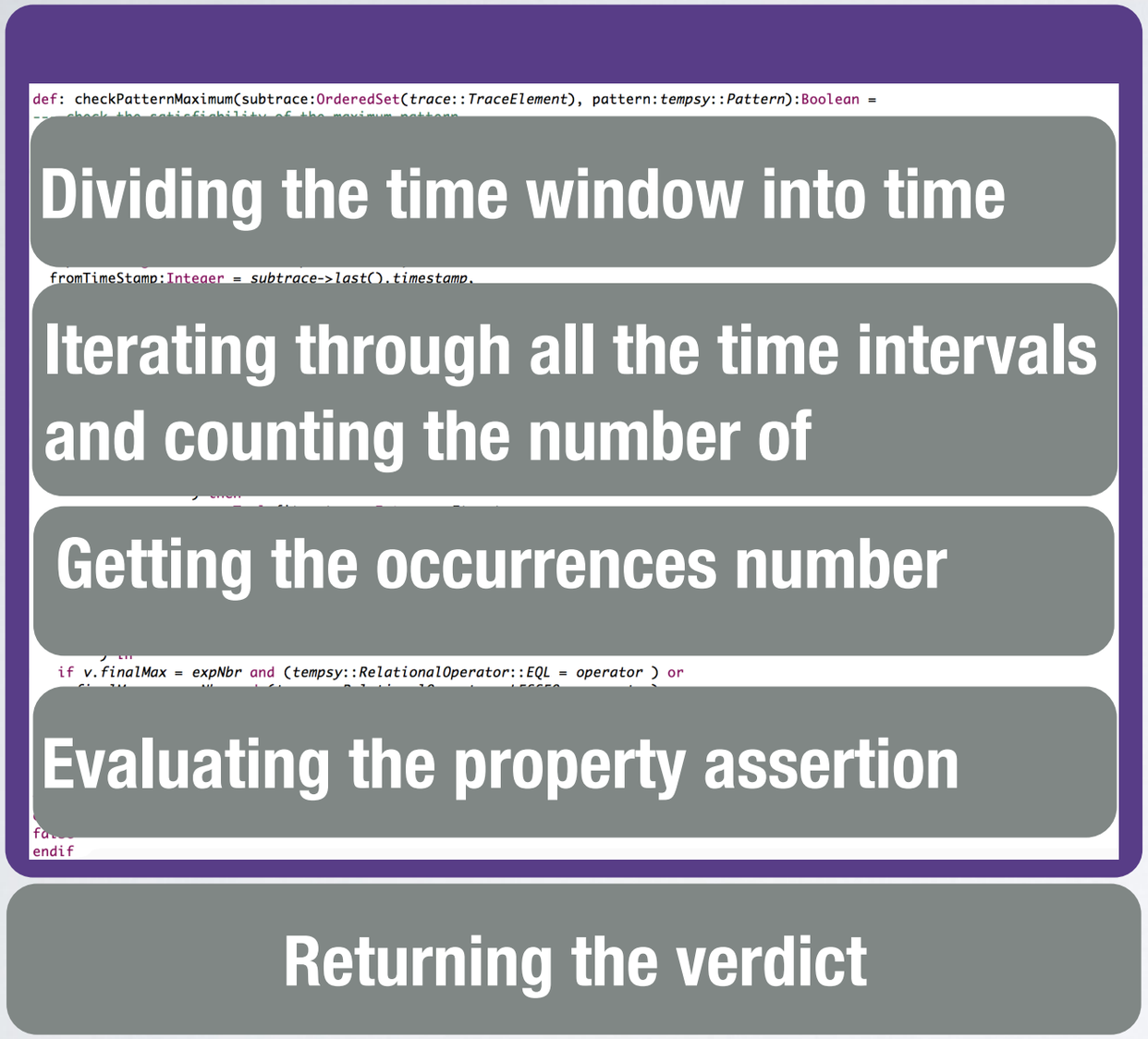
Returning the verdict

checkPatternMaximum



Returning the verdict

checkPatternAverage



Unlike the maximum pattern, the average pattern excludes the tail interval

checkPatternAverageElapsedTime

Iteration

If the timestamp of the current trace element is within the time window:

Find an occurrence of the first pair element

If the second pair element occurs:

Accumulate the distances between the pair

Keep track of the number of pair occurrences

Compute the average distances

Returning the verdict

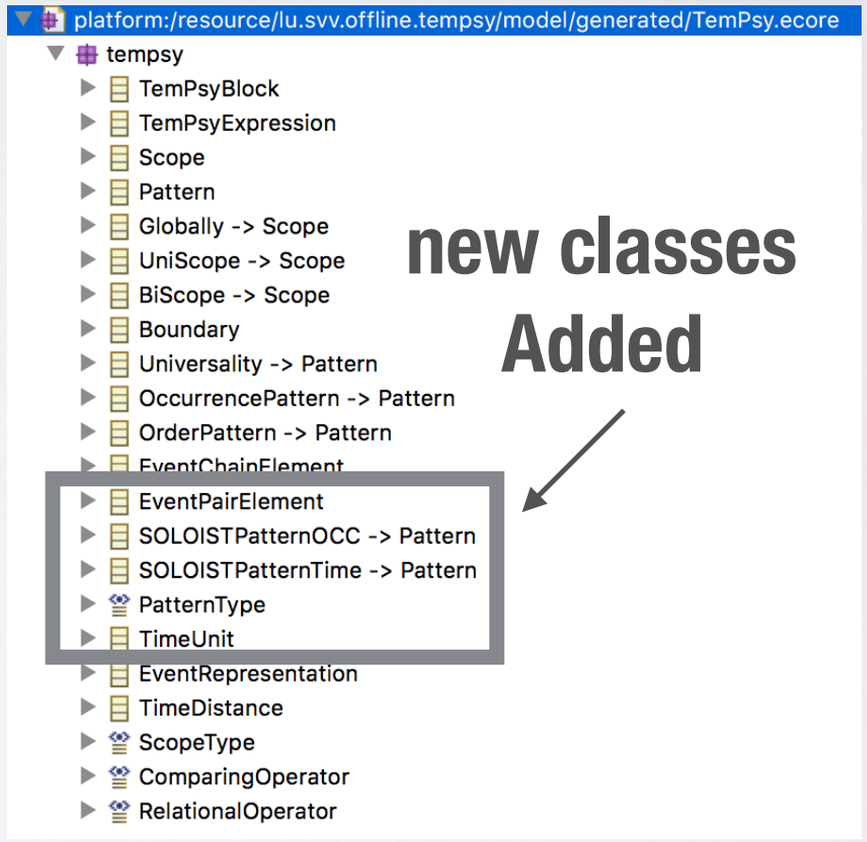
Implementation

TemPsy-AG in Xtext: The Case of the Maximum Pattern

Extended grammar: Xtext file

```
Pattern returns Pattern:

Universality | ExistencePattern | AbsencePattern | OrderPattern | MaximumPattern
AveragePattern | AverageElapsedTimePattern;

MaximumPattern returns SOLOISTPatternOCC:

patternType=MaximumPatternType
(event=EventRepresentation) 'within'
TimeUnit1= TimeUnits 'every'
TimeUnit2= TimeUnits
relationalOperator=RelationalOperator
expectedNumber= EInt;


enum MaximumPatternType returns PatternType:
SOLOISTMax= 'maximum';

enum RelationalOperator returns RelationalOperator:
GRTEQ = '>=' | GRT= '>' | LESSEQ= '<=' | LESS = '<' | EQL= '==' ;
```

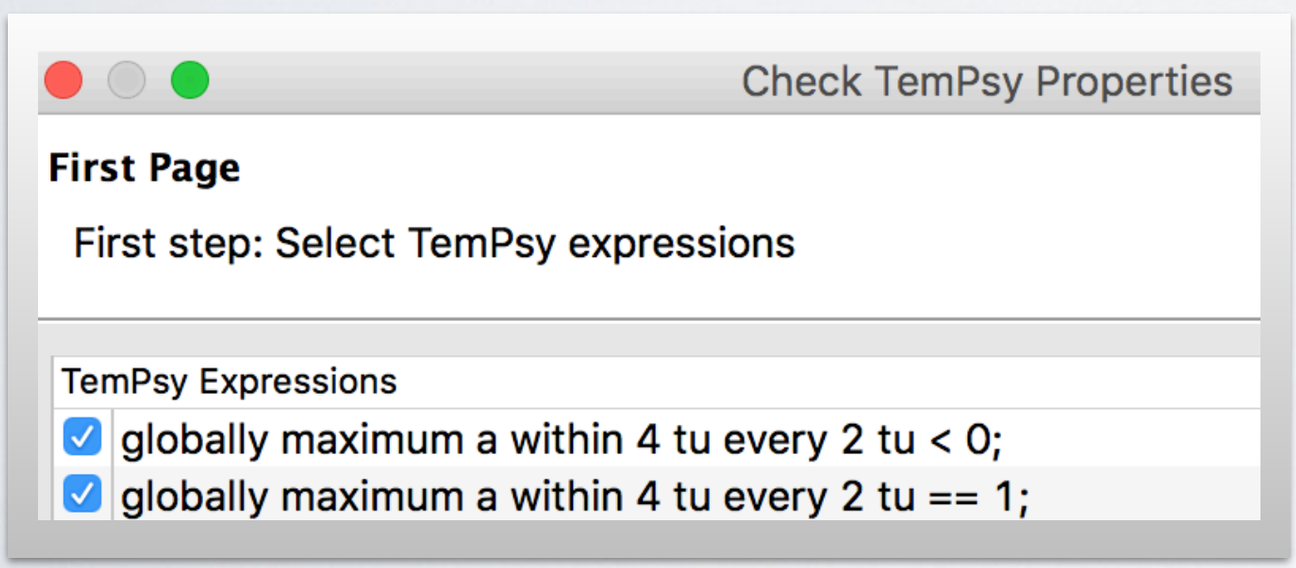
Extended TemPsy: ecore file



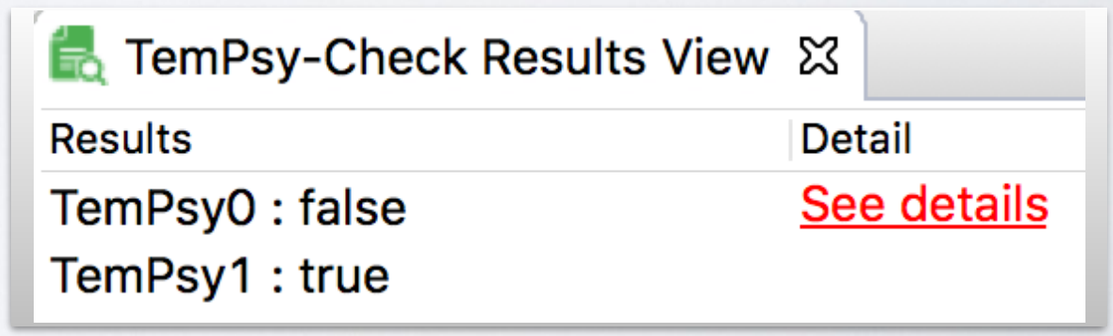
TemPsy-Check invocation in TemPsy-Suite

- TemPsy-Suite is an Eclipse plugin
- We extended the plugin to verify SOLOIST patterns

Loading properties with the maximum pattern in the editor



Invoking TemPsy-Check: TemPsy-Check results view



Evaluation

Research Questions:

- **RQ1:** How does the trace length in the three supported patterns affect the execution time of our approach?

- **RQ2:** How does the number of time intervals in the maximum and the average patterns affect the execution time of our approach?

- **RQ3:** How does TemPsy-AG compare to a state-of-the-art tool in terms of execution time?

Synthesized Traces for Evaluation

- Real traces are often inadequate to cover a large range of trace lengths and a variety of properties
- They guarantee a great diversity in terms of patterns occurrences in the traces
- They allow us to focus on the scalability of the approach

Trace Generation Strategy
Maximum pattern

- Given K and h, counting the number of time intervals which is equal to $\lfloor K / h \rfloor$

- Generating a specific number of event occurrences per time interval with respect to the relational operator.
  Example: For “< 5”, we generate a random number of occurrences x, so that x is in [0..4]

Trace Generation Strategy
Average pattern

- Number of event occurrences = number of time intervals * bound value

- Assigning the occurrences number to all the time intervals with respect to the bound value and the relational operator to get the desired average

Trace Generation Strategy
Average elapsed time pattern

- Generating a specific pair occurrences number of the events mentioned in the property

- Generating distances =
  bound value * events pair occurrences

- Assigning the distances number to all the event pair occurrences

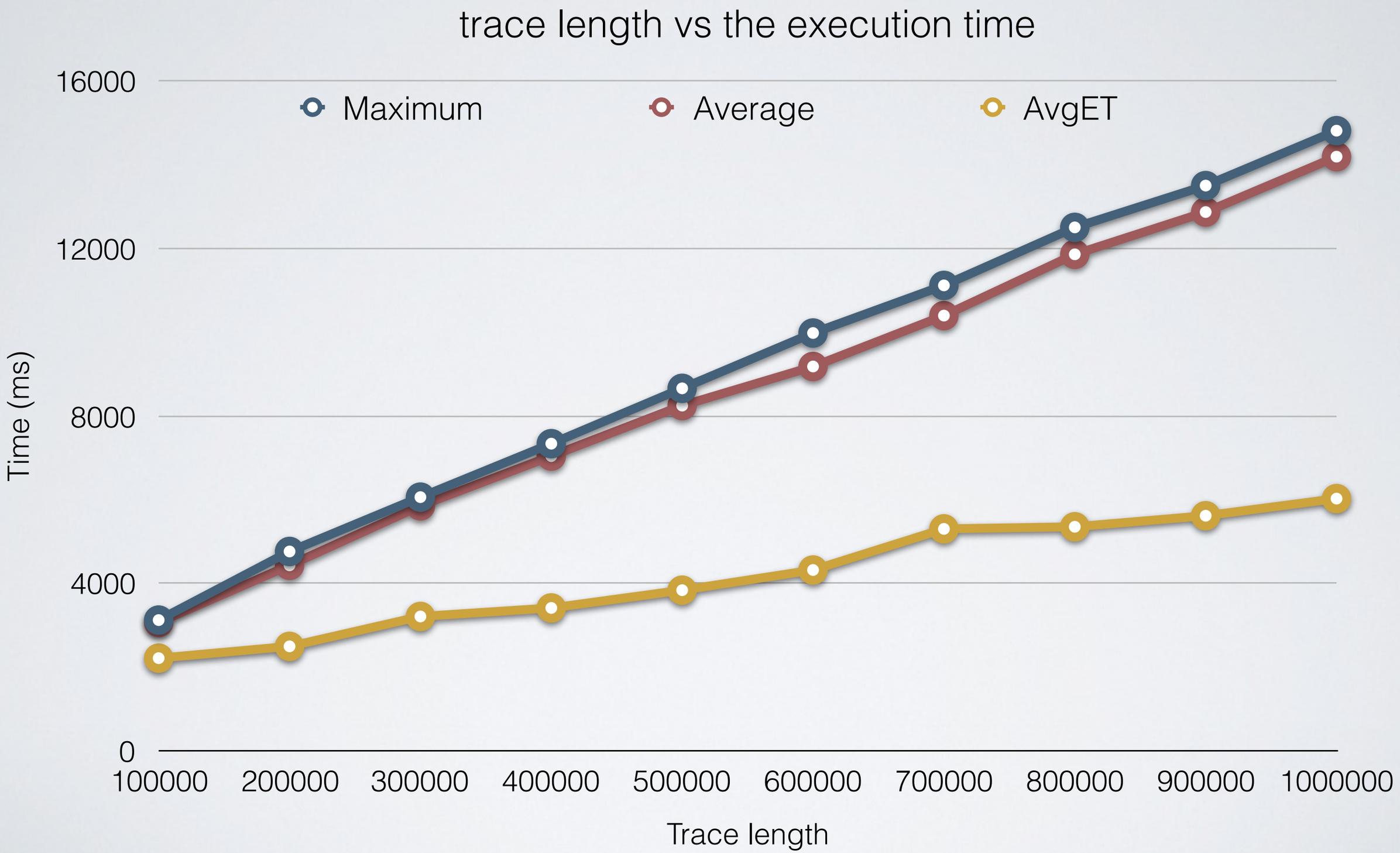
Properties used for the evaluation

- **maximum pattern:**
  - `globally maximum a within k tu every h tu <= 20 ;`
- **average pattern:**
  - `globally average a within k tu every h tu <= 20 ;`
- **average elapsed time pattern:**
  - `globally avgET for pair(s) (a,b) within k tu <= 20 ;`

parameters values of the properties for answering RQ1 (scalability in terms of trace length)

- K = trace length
- Considering 10 different trace lengths: 100K to 1M
- Fixing #Time intervals = 10 for each trace length

RQ1 (scalability wrt trace length)



parameters values of the properties for answering RQ2 (scalability wrt # of intervals)

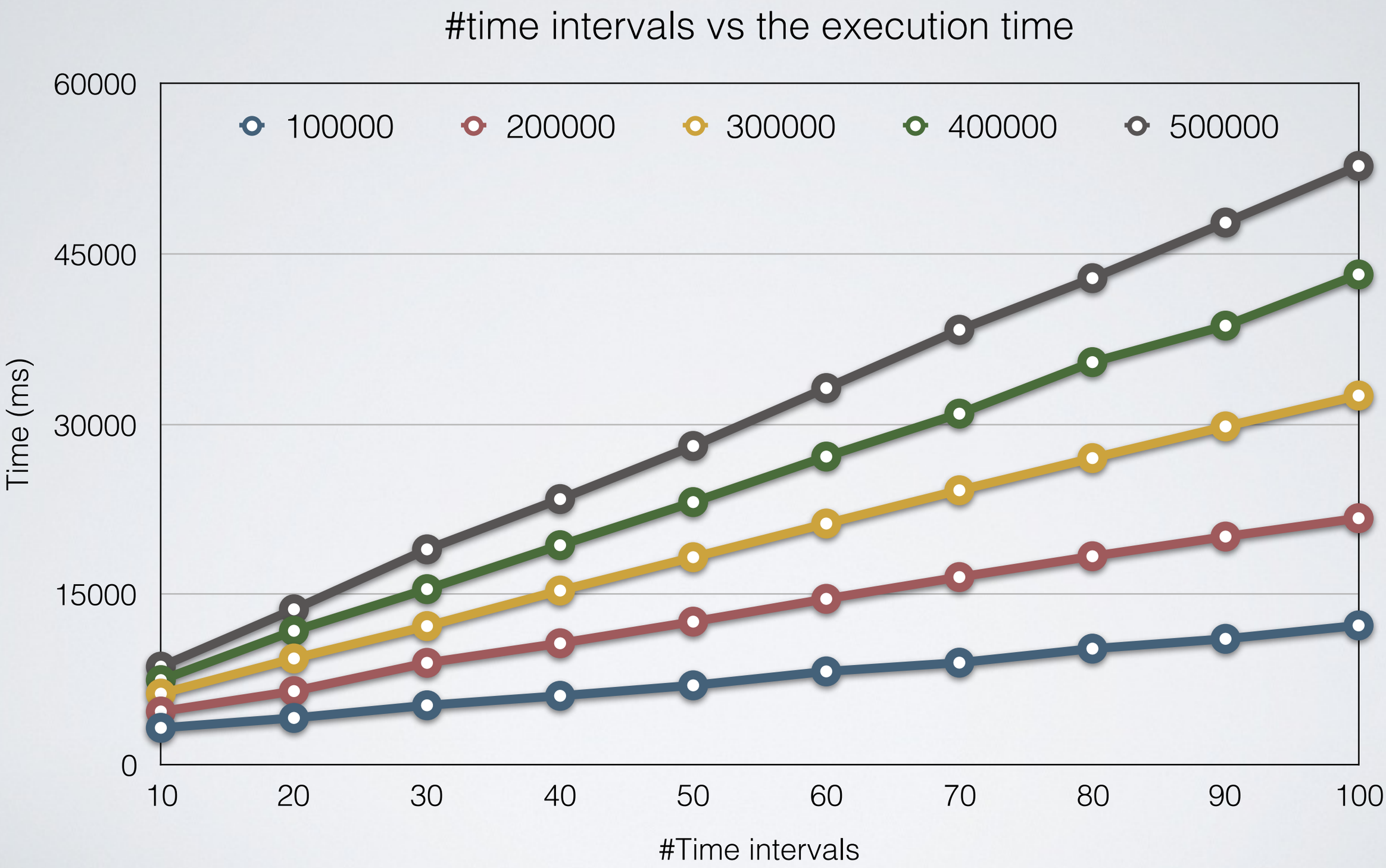
- K = trace length

- 5 different trace lengths: 100k to 500k

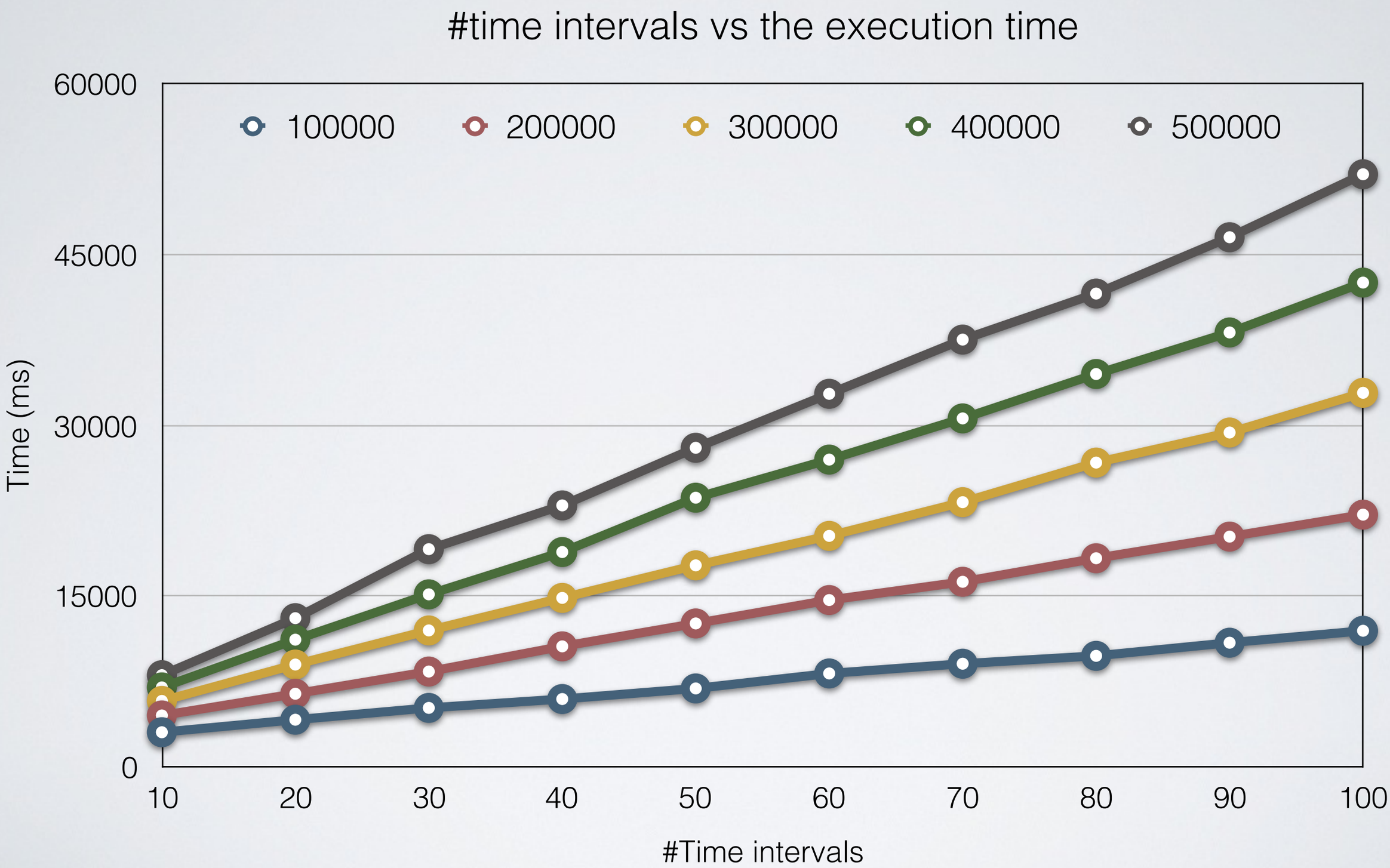
- Maximum/Average patterns: we variate h in such a way that we get #Time intervals in [10..100].
  Example: for trace length = 100k, #Time intervals = 10

➔ h= 10k

RQ2: Maximum Pattern



RQ2: Average Pattern



RQ3: Comparison of TemPsy-AG with SOLOIST-translator (1)

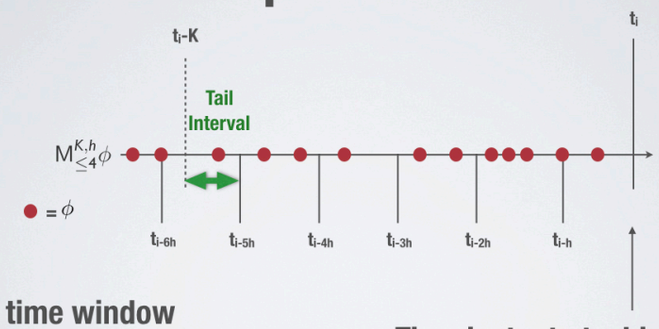
- **Baseline: SOLOIST-translator (state-of-the-art tool for SOLOIST)**
- **SOLOIST-translator went out of memory for all the generated traces whose length is equal to100k for: #Time intervals in [10..100]**
- **For very small instances: trace lengths variating from 500 to 2k, SOLOIST-translator crashed for the largest trace length**

RQ3: Comparison of TemPsy-AG with SOLOIST-translator (2)

#Time intervals / Trace length	10		50		100	
	TemPsy-AG	soloist-translator	TemPsy-AG	soloist-translator	TemPsy-AG	soloist-translator
500	1211	1376	1262	1814	1315	2346
1000	1249	4601	1341	5945	1463	7507
2000	1274	-	1469	-	1598	-

Summing up…

“Maximum number of events” pattern

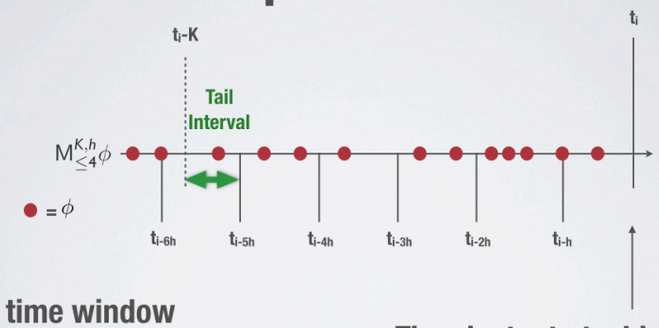


K= time window

h= time interval

Time instant at which the property is evaluated

“Maximum number of events” pattern



K= time window

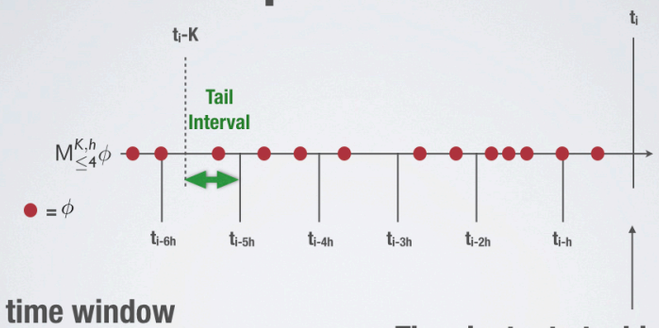
h= time interval

Time instant at which the property is evaluated

Motivations

- Requirements specifications based on temporal properties with aggregation operators are common
- The support in terms of verification of such properties is limited
  - The only language that supports such temporal properties is SOLOIST
    - The corresponding tool **does not scale** with respect to the **length of the trace!**

“Maximum number of events” pattern



K= time window

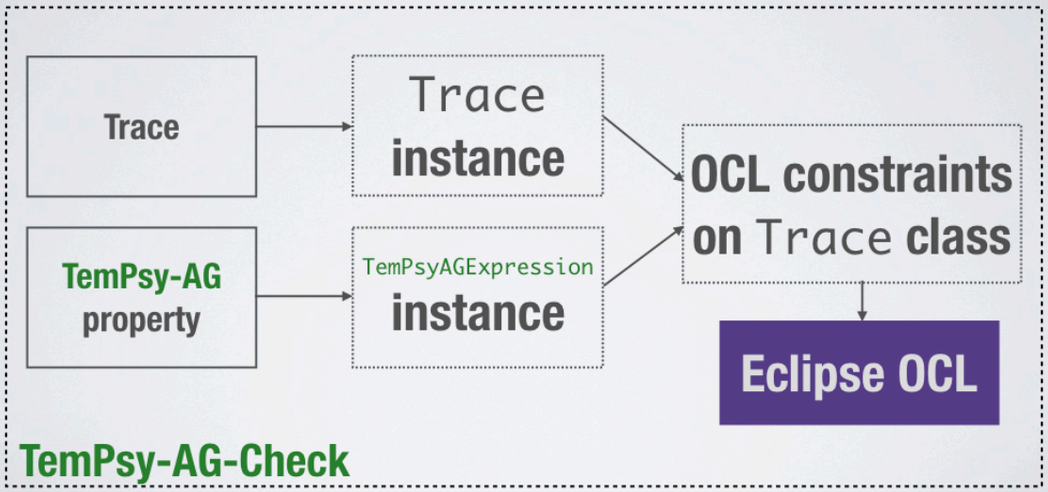
h= time interval

Time instant at which the property is evaluated

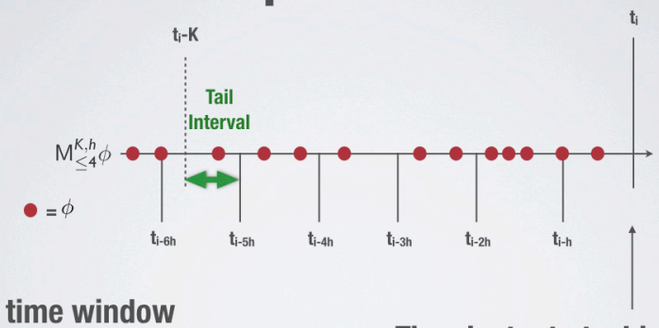
Motivations

- Requirements specifications based on temporal properties with aggregation operators are common
- The support in terms of verification of such properties is limited
  - The only language that supports such temporal properties is SOLOIST
    - The corresponding tool **does not scale** with respect to the **length of the trace!**

Model-driven Offline Trace Checking of TemPsy-AG Properties



"Maximum number of events" pattern



K= time window

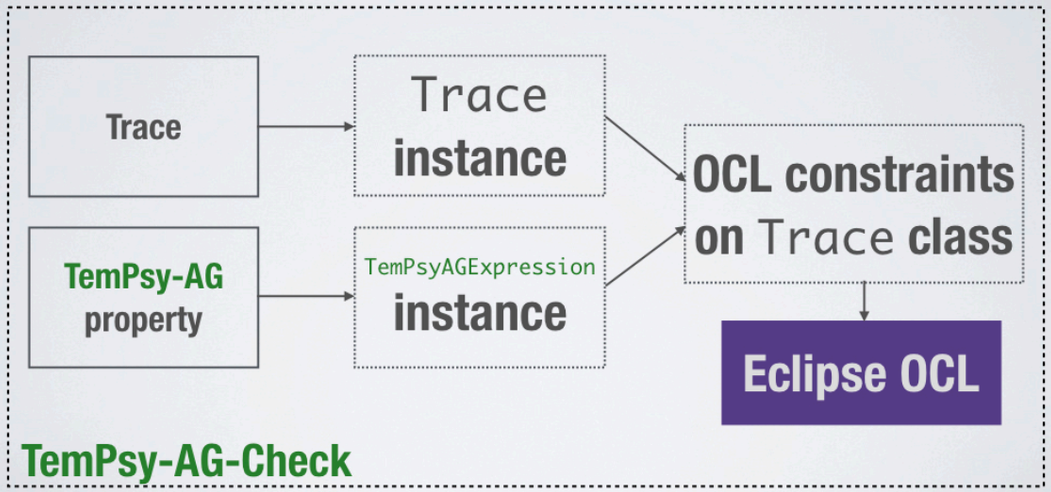
h= time interval

Time instant at which the property is evaluated

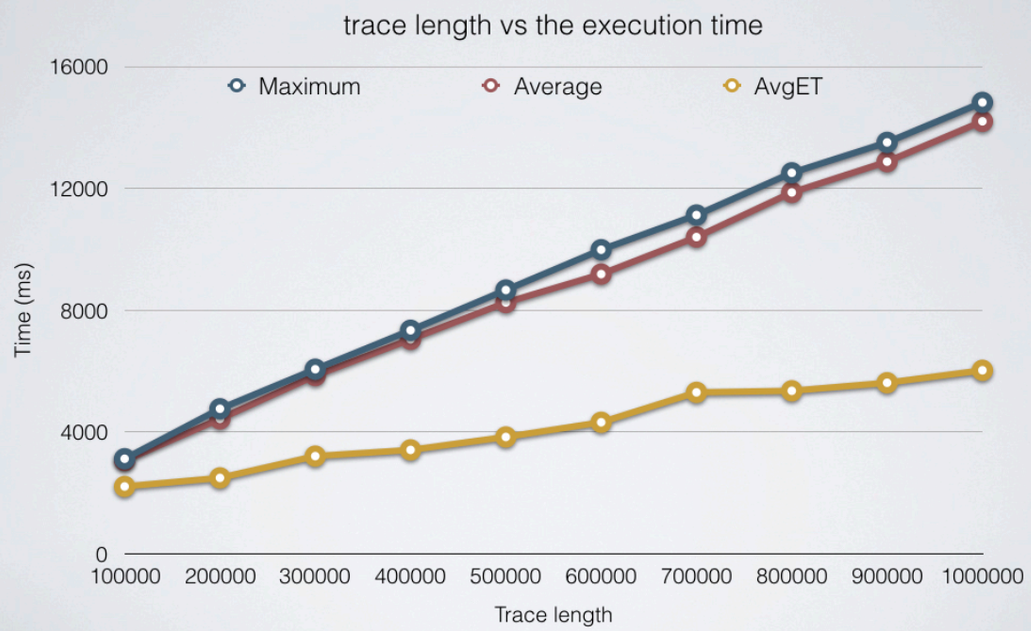
Motivations

- Requirements specifications based on temporal properties with aggregation operators are common
- The support in terms of verification of such properties is limited
  - The only language that supports such temporal properties is SOLOIST
    - The corresponding tool **does not scale** with respect to the **length of the trace!**

Model-driven Offline Trace Checking of TemPsy-AG Properties



RQ1 (scalability wrt trace length)



Conclusions

- Evidence of feasibility and viability of extending a model-driven runtime verification approach
- Adding support for a larger class of properties
- Retaining acceptable performance from a practical standpoint

Future work

- Applying our approach on real traces from a case study
- Working on distributed trace checking using big data technologies (e.g., MapReduce)

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

Chaima BOUFAIED, Domenico BIANCULLI, Lionel BRIAND

ECMFA'19