TemPsy-Check: a Tool for Model-driven Trace Checking of Pattern-based Temporal Properties

Wei Dou, Domenico Bianculli, and Lionel Briand

SnT Centre - University of Luxembourg, Luxembourg
{dou,bianculli,briand}@svv.lu

Abstract

TemPsy (Temporal Properties made easy) is a pattern-based, domain-specific language for the specification of temporal properties. In this paper we provide an overview of TemPsy-Check, a tool that implements a model-driven approach for performing offline trace checking of temporal properties written in TemPsy. TemPsy-Check relies on an optimized mapping of temporal requirements written in TemPsy into Object Constraint Language (OCL) constraints on a conceptual model of execution traces.

1 Introduction

The tool presented in this paper has been developed as part of a research collaborative project that we run 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 [8]. In this project we have investigated 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 context of this project set three main requirements for the development of the solution:

R1) when analysts do not have adequate skills to make use of temporal logic, an *alternative* domain-specific language should be provided to facilitate the specification of business process requirements;

R2) to be viable in the long term, any solution shall rely on standard and stable MDE (model-driven engineering) technology for checking the compliance of a system to its application requirements;

R3) any solution shall be scalable, such that a trace with millions of events could be checked within seconds.

To fulfill requirement R1, in previous work [7] 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. [10], 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) [6], 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.

To fulfill requirements R2 and R3, in previous work [9] we proposed a scalable model-driven trace checking procedure, which relies on a mapping of temporal requirements written in *TemPsy* into Object Constraint Language (OCL) [14] constraints on a conceptual model of execution traces. This mapping is optimized based on the structure of the *TemPsy* property to check, in order to achieve better performance.

In this paper we discuss the TEMPSY-CHECK tool[1], which implements the procedure presented in [9]. TEMPSY-CHECK is available at http://weidou.github.io/TemPsy-Check/.

2 The *TemPsy* language

TemPsy [6] is the most recent version of our previous proposal [7] for a pattern-based, domain-specific specification language for temporal properties; it is based on an extended version of the catalogue of property specification patterns defined in [10].

The design of *TemPsy* has been driven by the analysis of the requirements documentation of various applications developed as business processes by our partner. This analysis revealed that the vast majority of these requirements could be expressed as temporal properties, enriched with timing information. More specifically, we were able to recast most of specifications written in natural language using the system of property specification patterns of Dwyer et al. In some cases, we extended the original definitions proposed in [10] to match the specifications; these extensions are:

1) The possibility, in the definition of a scope boundary, to refer to a specific occurrence of an event, as in "before the second occurrence of event X...". In the original definition of the pattern systems, boundaries of scopes refer implicitly to the first occurrence of an event.

2) The possibility to indicate a time distance with respect to a scope boundary, as in "at least two time units before the n-th occurrence of event X...".

3) Support for expressing time distance between events occurrences in the precedence and response patterns as well as in their chain versions, for expressing properties such as "event B should occur in response to event A within 2 time units".

4) Additional variants for the bounded existence and absence patterns.

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 to express temporal requirements commonly found in business process applications. *TemPsy* has received positive feedback from our partner, which has deemed it as suitable communication mechanism to express the requirements specifications of business processes. Our partner has integrated *TemPsy* into the SoftwareAG ARIS modeling tool [15], and its analysts have started using it to annotate business process models with *TemPsy* specifications.

2.1 Syntax

The syntax of *TemPsy* is shown in Fig. 1: non-terminals are enclosed in angle brackets, terminals are enclosed in single quotes, optional elements are enclosed in brackets, the character '+' indicates one or more occurrences of an element, the character '*' indicates zero or more occurrences of an element.

A *TemPsy* property, which is a denoted by the non-terminal ⟨*TemPsyBlock*⟩, comprises a set of ⟨*TemPsyExpression*⟩s combined by conjunction. Each *TemPsy* expression starts with an optional '`temporal`' keyword and has an optional alphanumeric identifier, followed by a ⟨*Scope*⟩ and a ⟨*Pattern*⟩. A ⟨*Scope*⟩ indicates the segment(s) of an execution trace in which a ⟨*Pattern*⟩ should hold.

The keywords indicating the five ⟨*Scope*⟩s identify univocally the corresponding scopes from [10] ('`globally`', '`before`', '`after`', '`between`'-'`and`', '`after`'-'`until`'). As for the ⟨*Pattern*⟩s,

[1] TEMPSY-CHECK can be considered a profound and optimized revision of its predecessor OCLR-CHECK, which was based on *OCLR* and participated in the 2nd Competition on Runtime Verification in 2015 [13].

⟨*TemPsyBlock*⟩	::=	⟨*TemPsyExpression*⟩+
⟨*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*⟩
⟨*Id*⟩	::=	⟨*IdStartChar*⟩ ⟨*IdChar*⟩*
	|	⟨*Id*⟩ (⟨*IdConnector*⟩ ⟨*Id*⟩)*
⟨*IdStartChar*⟩	::=	[A-Z] | ‘`_`’ | [a-z]
⟨*IdChar*⟩	::=	⟨*IdStartChar*⟩ | [0-9]
⟨*IdConnector*⟩	::=	‘`.`’ | ‘`::`’
⟨*Int*⟩	::=	[1-9] ([0-9])*

Figure 1: Syntax of *TemPsy*

‘`always`’ corresponds to universality, ‘`eventually`’ to existence, ‘`never`’ to absence, ‘`preceding`’ to precedence and precedence chain, ‘`responding`’ to response and response chain.

The definition of ⟨*Scope*⟩s and ⟨*Pattern*⟩s refers to the concept of ⟨*Event*⟩. We assume that an ⟨*Event*⟩ is represented by an alphanumeric string, to match the event names logged in the execution trace on which the properties specified in *TemPsy* are meant to be checked. ⟨*Scope*⟩s contain boundaries (expressed with ⟨*Boundary1*⟩ or ⟨*Boundary2*⟩) that denote a specific occurrence of an event as a boundary, possibly with a time distance; notice that ⟨*Boundary2*⟩ represents a syntactic restriction of ⟨*Boundary1*⟩. Chains of events, used in *precedence* and *response* patterns, are defined as ⟨*EventChainExp*⟩, which denotes a comma-separated list of events, possibly with a time distance (⟨*TimeDistanceExp*⟩) between each pair of events (denoted with the ‘`#`’ symbol). Time distances are expressed with an integer value, followed by the ‘`tu`’ keyword, which represents a generic time unit (i.e., any denomination of time).

2.2 *TemPsy* at Work

We now present some examples of properties that can be expressed with *TemPsy*, in order to provide the reader with a high-level, intuitive understanding of the language. We consider the

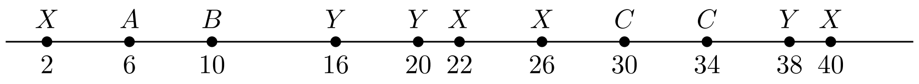

Figure 2: An event trace on which to evaluate the properties described in section 2.2; events are above the line, timestamps below

execution trace shown in Fig. 2; for each property[2] we first define it in English, indicate whether it is violated or not by the trace, and then express it in *TemPsy*.

p1) "Event C shall happen 8 time units after the second occurrence of event X." (satisfied)

`temporal p1: after 2` X `exactly 8 tu eventually` C

p2) "Event A shall happen within 30 time units after the first occurrence of event X." (satisfied)

`temporal p2: after` X `at most 30 tu eventually` A

p3) "Event C shall eventually happen after at least 3 time units since the first occurrence of event X; and it shall happen before event Y if the latter happens." (violated because event C occurs after event Y)

`temporal p3: after 1` X `at least 3 tu until` Y `eventually` C

p4) "After the second occurrence of event X, event C shall eventually happen exactly twice." (satisfied)

`temporal p4: after 2` X `eventually exactly 2` C

p5) "Event C shall happen at least once between every first occurrence of event X and the next event Y; the time interval between event X and the first occurrence of event C shall be at least 5 time units." (violated because event C does not occur between the first segment delimited by event X on the left and event Y on the right)

`temporal p5: between` X `at least 5 tu and` Y `eventually at least 1` C

p6) "Event B shall happen at least 3 time units before the first occurrence of event Y." (satisfied)

`temporal p6: before` Y `at least 3 tu eventually` B

p7) "Before the first occurrence of event Y, once event X occurs, event A shall happen followed by event B; the time interval between X and A shall be at least 3 time units." (satisfied)

`temporal p7: before` Y A`,` B `responding at least 3 tu` X

The informal and the formal semantics are described in the online technical report [6].

3 Model-driven Trace Checking with *TemPsy*

The idea at the basis of the model-driven trace checking approach implemented by *TemPsy* is to *reduce* the problem of checking a *TemPsy* property ρ over a trace λ, to the problem of

[2]These properties are given as an example and should be considered individually, rather than together as a set; they do not correspond to the specification of a real system.

evaluating an OCL constraint (semantically equivalent to ρ) on an instance of a conceptual model for execution traces (equivalent to λ).

This reduction allows us to rely on standard and stable MDE technology to perform offline trace checking. Indeed, standard OCL checkers, such as Eclipse OCL [11], can be used to evaluate OCL constraints on model instances. The use of a model-driven approach and of standard technologies fulfills requirement R2 stated in section 1, and enables us to provide a practical and scalable solution for trace checking of temporal properties, which is also viable in the long term.

At the basis of this approach there is the definition of a conceptual model for execution traces, since the transformation of *TemPsy* properties into efficiently checkable *OCL constraints defined on such model* is a key strategy for us to achieve scalability. The model, not depicted here for space reasons, contains a `Trace`, which is composed of a sequence of `TraceElements`, accessed through the association `traceElements`. Each `TraceElement` contains an attribute `event` of type string, which represents the actual event recorded in the trace, and an attribute `timestamp` of type integer, which indicates the time at which the event occurred.

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

TEMPSY-CHECK, when given in input a set of properties to check and a trace file, creates an instance of the `Trace` class based on the trace input file.

The key step of our approach is to evaluate an OCL invariant on this trace instance for every *TemPsy* property provided in input. The checking of this invariant, which can be done using standard OCL checking tools, is semantically equivalent to performing trace checking of the *TemPsy* property. The invariant is roughly equivalent to this OCL expression:

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

Notice that in the actual constraints, the placeholder `*S*` is replaced with a string from {`Globally`, `Before`, `After`, `BetweenAnd`, `AfterUntil`} and the placeholder `*P*` is replaced with a string from {`Universality`, `Existence`, `Absence`, `Precedence`, `Response`}. These strings correspond, respectively, to the scope and the pattern used in the input *TemPsy* property to check. In the invariant, the variable `subtraces` contains the portion(s) of the trace returned by the function `applyScope*S*`, which takes an instance of a *TemPsy* `Scope` as input parameter. The invariant checks, by calling the function `checkPattern*P*`, whether the input `pattern` holds on each sub-trace in `subtraces`.

The details of the procedure implemented by TEMPSY-CHECK and the definition of the various OCL functions of type `applyScope*S*` and `checkPattern*P*` are available in [9, 6].

4 Implementation

The implementation of TEMPSY-CHECK is based on Xtext [12] and Eclipse OCL [11]. The tool takes as input a list of *TemPsy* expressions represented in an XMI-based format and a trace instance in CSV format. Any *TemPsy* expressions defined in the textual notation shown in Fig. 1 can be converted to the XMI-based format using another tool included in the distribution. TemPsy-Check provides file readers for loading *TemPsy* expressions and trace instances; it can be extended to support other formats thanks to its design following the *Strategy Pattern*. Moreover, we have developed a Java class `ConstraintFactory` to help build OCL constraints corresponding to the input *TemPsy* expressions. The evaluation of the OCL invariants is done using the OCL checker included in Eclipse. The boolean output of the checker is then returned to the user.

5 Summary of Experimental Results

To fulfill requirement R3, we extensively evaluated the scalability of TemPsy-Check by assessing the relationship among the checking time, the structural properties of a trace (e.g., length, distribution of events), and the type of property to check; we used real properties extracted from a case study developed in collaboration with our partner CTIE, on traces with length ranging from 100K to 1M. We also compared the performance of TemPsy-Check with MonPoly [3], a state-of-the-art alternative technology, selected from the participants to the "offline monitoring" track of the international Competition on Software for Runtime Verification [2, 13].

The experimental results show that TemPsy-Check can load and analyze very large traces (with one million events) in about two seconds and that it scales linearly with respect to the length of the trace to check. The results also show that TemPsy-Check in practice performs similarly to or better than[3] the state-of-the-art, depending on the type of properties, confirming the feasibility and benefits of a model-driven approach for trace checking of temporal properties.

The detailed evaluation methodology and the complete evaluation data are available in [9, 6].

6 Discussion and Future Work

In this paper we have provided an overview of our tool TemPsy-Check, a practical and scalable solution for trace checking of pattern-based temporal properties written in *TemPsy*. TemPsy-Check relies on an efficient mapping of requirements written in *TemPsy* into regular OCL constraints on a conceptual model for execution traces. We have applied TemPsy-Check for the verification of real properties derived from a case study of our public service partner in the context of eGovernment business process modeling. More in general, TemPsy-Check can be used in contexts where model-driven engineering is already a practice and where relying on standards and industry-strength tools for property checking is a fundamental prerequisite.

As part of future work, we plan to extend TemPsy-Check to provide a more informative output than the boolean result currently returned when violations are detected in a trace, by adding support for interactive inspection of violations. We also plan to extend *TemPsy* and TemPsy-Check to support the service provisioning specification patterns introduced in [4] and implemented in the SOLOIST language [5].

Acknowledgment. This work has been partially supported by the National Research Fund, Luxembourg (FNR/P10/03).

[3] We remark that the specification language of MonPoly (MFOTL) is more expressive than *TemPsy* (e.g., by supporting first-order quantification), hence the performance of MonPoly could have been negatively affected by the more complex implementation needed to support a richer specification language.

References

[1] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. *IEEE Trans. Softw. Eng.*, 41(7):620–638, 2015.

[2] Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. First international competition on software for runtime verification. In *Proc. RV 2014*, volume 8734 of *LNCS*, pages 1–9. Springer, Heidelberg, Germany, 2014.

[3] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. MONPOLY: Monitoring usage-control policies. In *Proc. RV 2011*, volume 7186 of *LNCS*, pages 360–364, Heidelberg, Germany, 2012. Springer.

[4] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *Proc. ICSE 2012*, pages 968–976, Piscataway, NJ, USA, 2012. IEEE.

[5] Domenico Bianculli, Carlo Ghezzi, and Pierluigi San Pietro. The tale of SOLOIST: a specification language for service compositions interactions. In *Proc. FACS'12*, volume 7684 of *LNCS*, pages 55–72, Heidelberg, Germany, 2013. Springer.

[6] Wei Dou, Domenico Bianculli, and Lionel Briand. A model-based approach to offline trace checking of temporal properties with OCL. Technical Report TR-SnT-2014-5, SnT Centre - University of Luxembourg, September 2014. http://hdl.handle.net/10993/16112.

[7] Wei Dou, Domenico Bianculli, and Lionel Briand. OCLR: a more expressive, pattern-based temporal extension of OCL. In *Proceedings of the 2014 European Conference on Modelling Foundations and Applications (ECMFA 2014), York, United Kingdom*, volume 8569 of *Lecture Notes in Computer Science*, pages 51–66. Springer, July 2014.

[8] Wei Dou, Domenico Bianculli, and Lionel Briand. Revisiting model-driven engineering for run-time verification of business processes. In *Proc. SAM 2014*, volume 8769 of *LNCS*, pages 190–197, Heidelberg, Germany, September 2014. Springer.

[9] Wei Dou, Domenico Bianculli, and Lionel Briand. A model-driven approach to trace checking of pattern-based temporal properties. In *Proceedings of the 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA*. IEEE, September 2017.

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

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

[12] Eclipse. Xtext–Language Engineering Made Easy! http://www.eclipse.org/Xtext/, November 2015.

[13] Yliès Falcone, Dejan Ničković, Giles Reger, and Daniel Thoma. Second international competition on runtime verification. In *Proc. RV 2015*, pages 405–422, Heidelberg, Germany, 2015. Springer.

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

[15] Software AG. ARIS. http://www.softwareag.com/corporate/products/aris/default.asp, 2014.