securityandtrust.lu

Software Verification and Validation Laboratory: A Model-Driven Approach to Offline Trace Checking of Temporal Properties with OCL

Wei Dou, Domenico Bianculli and Lionel Briand

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

TR-SnT-2014-5

ISBN: 978-2-87971-125-6

Last update: March 31, 2016

Published on: March 20, 2014

UNIVERSITÉ DU LUXEMBOURG

A Model-Driven Approach to Offline Trace Checking of Temporal Properties with OCL

Wei Dou, Domenico Bianculli, and Lionel Briand

Abstract

Offline trace checking is a procedure for evaluating requirements over a log of events produced by a system. The goal of this paper is to present a practical and scalable solution for the offline checking of the temporal requirements of a system, which can be used in contexts where model-driven engineering is already a practice, where temporal specifications should be written in a domain-specific language not requiring a strong mathematical background, and where relying on standards and industry-strength tools for property checking is a fundamental prerequisite. The main contributions are: the *TemPsy* language, a domain-specific specification language based on common property specification patterns, and extended with new constructs; a model-driven offline trace checking procedure based on the mapping of requirements written in *TemPsy* into OCL (Object Constraint Language) constraints on a conceptual model on execution traces, which can be evaluated using an OCL checker; the implementation of this trace checking procedure in the TEMPSY-CHECK tool; the evaluation of the scalability of TEMPSY-CHECK and its comparison to a state-of-the-art alternative technology. The proposed approach has been applied to a case study developed in collaboration with a public service organization, active in the domain of business process modeling for eGovernment.

Index terms— Trace checking, temporal properties, property specification patterns, model-driven engineering, OCL

1 Introduction

Modern enterprise information systems are often designed and built using the principles and technologies of business process modeling, based on business process languages like BPMN (Business Process Model and Notation) [53]. Recently, the design and implementation of business processes have started leveraging model-driven engineering (MDE) methodologies [19] and code generation techniques. For example, our public service partner CTIE (Centre des technologies de l'information de l'Etat, the Luxembourg national center for information technology[1]), from which we draw the main motivation of this work and our case study, has developed in-house a model-driven methodology for designing eGovernment business processes.

These business processes are usually very complex and are realized as compositions of services provided by different administrations, and third-party suppliers. They act as the "glue" to orchestrate different information systems, possibly by many different organizations, in an effort to foster cooperation of various administrations. Designing and operating effective and efficient processes to drive e-service delivery is one of the most challenging tasks for public administrations. The correct enactment of business processes is of utmost importance to guarantee reliable digital solutions to citizens and enterprises, as well as to foster an effective cooperation of the various public administrations in a state.

From a more general standpoint, in information systems, the correct enactment of a business process can be ensured [4] by: 1) precisely specifying its requirements; 2) using a verification technique to check the compliance of the business process with respect to its requirements.

Regarding the specification of requirements of business processes, the analysis of the requirements of various applications developed as business processes by our partner revealed that the majority of these requirements could be expressed as temporal constraints, enriched with timing information. Examples of these properties are constraints on the sequence and number of occurrences of events, with additional constraints on the temporal distance between events. This type of properties has been widely studied in the context of concurrent, real-time critical systems [27] and, more recently, also in other domains like service-based applications [15,41,48,58] and automotive [55]. There have been several proposals to formally specify these properties; many of these proposals rely on some temporal logic, either the classic LTL or CTL, or more specialized versions like SOLOIST [16]. However, the problem in using these specification approaches is twofold: 1) they require strong theoretical and mathematical background, which are rarely found among practitioners; 2) the support in terms of verification tools is limited and often based on prototypes that do not scale for industrial applications. To partially mitigate the first problem, researchers have proposed catalogues of *property specification patterns* [2, 15, 27, 38, 43], which collect generalized, proven solutions for expressing recurrent, common types of specifications. In some cases, catalogues include a restricted natural language grammar front-end to express the patterns, and a mapping of the semantics of (restricted) natural language constructs to temporal logic formalisms; this mapping can be automated with tools like PSPWizard [49]. While property specification patterns can make the formal specification of requirements easier, their concrete application results in the generation of a specification in a temporal logic, which leaves the second issue mentioned above still open. From the MDE side of specification languages there is OCL [54]. Although also based on mathematical foundations such as first-order logic and set theory, OCL includes many helper functions—to keep the constraints compact—and navigation expressions that reflect the structure of class diagrams (conceptual models)—to help with writing expressions that look more alike to program code. These fea-

[1] `www.ctie.public.lu`.

tures made OCL the de-facto constraints specification language in MDE practice and an international standard [54], which is supported by mature constraint checking technology, such as the constraint/query evaluator included in Eclipse OCL [28]. However, OCL does not support natively the specification of temporal constraints in an intuitive fashion. To overcome this limitation, several temporal extensions of OCL have been proposed in the literature [18, 23, 33, 46, 59, 62]; however, these extensions include temporal logic operators and thus intrinsically inherit the limitations of other specification approaches based on temporal logic. Other temporal extensions of OCL, such as [34, 42, 45, 57], explicitly support property specification patterns. Nevertheless, these pattern-based temporal extensions of OCL have limited expressiveness. For example, based on our analysis of a case study in eGovernment systems, none of the current pattern-based temporal extensions of OCL could support a property like "If the physical information of the card requester is collected within three days after the second approval notification, the card will be produced and then issued to the requester", which contains a reference to a specific occurrence of an event ("after the second approval notification ...") as well as an explicit temporal distance from an event ("... within three days...").

As for the second step towards the correct enactment of business processes, the compliance of a business process with respect to its requirements can be checked with different verification techniques, such as model checking [17,35], run-time monitoring [3, 41, 56, 58], and offline trace checking [13]; in this work we focus on the latter. Offline trace checking, also called *trace validation* [51] or *history checking* [31], is a procedure for evaluating requirements (usually specified in a temporal logic) over a log of recorded events produced by a system. Traces can be produced at run time by a proper monitoring/logging infrastructure, and made available at the end of a business process execution to perform offline trace checking. Offline trace checking complements verification activities performed before the deployment of a system, by allowing for the post-mortem analysis of actual behaviors emerged at run time and recorded on a log. These behaviors include the ones of the business process as well as those derived from the interaction of the business process with the various third-parties (e.g., other administrations, suppliers) involved in the execution of the process itself. Offline trace checking is thus also a way to check whether third-party providers fulfill their guarantees and to assess how they interact with the rest of the parties involved in the business process.

The goal of this paper is to present a practical and scalable solution for the offline checking of the temporal requirements of a business process, which is expected to be advantageous in contexts where the following requirements hold: 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 technology for checking the compliance of a business process to the application requirements; R3) any solution shall be scalable, such that a trace with millions of events could be checked within seconds. This goal is motivated by specific requirements from our partner in the context of business process models for eGovernment systems. Nevertheless, we believe, based on experience, that these requirements can be generalized to other contexts in which analysts cannot handle the mathematical background required by temporal logic and solutions have to be engineered by using MDE technologies already in place in the targeted development environment.

To achieve the above objectives, the paper will make the following contributions: i) the *TemPsy* (Temporal Properties made easy) language, a pattern-based domain-specific language for the specification of temporal properties; ii) a model-driven trace checking procedure, which relies on a mapping of temporal requirements written in *TemPsy* into OCL constraints on a conceptual model of execution traces; iii) a publicly available tool (TemPsy-Check) implementing this model-driven trace checking procedure; iv) an evaluation of the scalability of TemPsy-Check, applied to the verification of real properties derived from a case study of our public service partner, including a comparison with a state-of-the-art alternative technology. As a separate contribution, we also make available the artifacts used in the evaluation to contribute to the building of a public repository of case studies for evaluating trace checking/run-time verification procedures.

TemPsy[2] is a domain-specific language for the specification of temporal properties based on the catalogue of property specification patterns defined by Dwyer et al. [27] (with some extensions). To fulfill requirement R1 above, based on the discussions with our partner business analysts, we decided that the language should have the following features: be as close to natural language as possible, make no use of mathematical constructs, and support the commonly understood concepts used in the specification of requirements in the domain of business process modeling. Regarding the latter feature, we analyzed the requirements specifications of our industrial case study, to understand the type of specifications written (in natural language) by business analysts and to characterize them in terms of the property specification patterns in [27] (with some extensions). The relevant concepts and patterns found through this analysis drove the design of *TemPsy*, which resulted in a language sporting a syntax close to natural language, with all the constructs required to express the property specification patterns found in our case study, and 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—in an intuitive way—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 [60], and its analysts have started using it to annotate business process models with *TemPsy* specifications. In this paper, we show the application of *TemPsy* for the specification of an excerpt of a business process extracted from the case study developed with our partner.

[2] The language can be viewed as a profound revision of our previous proposal [25].

Our offline trace checking procedure fulfills requirement R2 above since it follows a model-driven approach, based on industry-strength OCL checkers. The procedure relies on a generic conceptual model of system execution traces and leverages a mapping of *TemPsy* properties into OCL constraints defined over this trace model. This mapping is optimized based on the structure of the *TemPsy* property to check, in order to achieve better performance. More specifically, we show how the problem of checking a *TemPsy* property over an execution trace (i.e., the *TemPsy* trace checking problem) can be reduced to evaluating an OCL constraint (derived from the *TemPsy* property to check and semantically-equivalent to it) on an instance of the trace model; this check can be executed using standard OCL checkers.

To show the fulfillment of requirement R3 above, we extensively evaluated the scalability of the proposed offline trace checking procedure, by assessing the relationship among the checking time, the structural properties of a trace (e.g., length, distribution of events), and the type of property to check. We evaluated the scalability of our TemPsy-Check tool on 38 properties extracted from our case study, on traces with length ranging from 100K to 1M. We also compared the performance of TemPsy-Check with a state-of-the-art alternative technology, selected from the participants to the "offline monitoring" track of the first international Competition on Software for Runtime Verification [8] (CSRV 2014). The experimental results show that TemPsy-Check can 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 compares favorably with the state-of-the-art.

The rest of the paper is structured as follows. Section 2 provides some background concepts. In section 3 we introduce *TemPsy*, presenting its syntax and its (informal) semantics. In section 4 we show the application of *TemPsy* in a case study in the domain of eGovernment. Section 5 presents the formal semantics of *TemPsy*. Section 6 describes our model-driven approach for trace checking of *TemPsy* properties. Section 7 reports on the evaluation conducted with TemPsy-Check. Section 8 discusses related work. Section 9 concludes the paper, providing directions for future work.

2 Background: Property Specification Patterns

A *pattern* represents a reusable solution for a recurrent problem [1]. Though initially proposed in the context of architecture [1], this concept has been adopted also in different sub-domains of software engineering, including software design, with design patterns [36], and formal verification, with *property specification patterns* [2].

Property specification patterns have been initially proposed by Dwyer et al. [27] in the late '90s in the context of formal verification, as a means to express recurring properties in a generalized form, which could be formalized in different specification languages, such as temporal logic. The goal of property specification patterns is to facilitate the writing of formal specifications, which can then be used with formal verification tools (e.g., model checkers).

Several catalogues of property specification patterns have been proposed in the literature [15, 27, 38, 39, 43]. In the rest of this section we provide a brief overview of the catalogue of property specification patterns by Dwyer et al. [27], which have been included (with some extensions) in the definition of the *TemPsy* language.

This catalogue[3] contains nine parametrizable *patterns*, representing high-level abstractions of formal specifications, and five *scopes*, which indicate the portions of a system execution in which a certain pattern should hold. In the following, we use the letters W, X, Y, and Z, to denote events or states of a system execution The five scopes, depicted in Fig. 1, are:

Globally. This scope corresponds to the entire system execution (i.e., the entire trace).

Before. It identifies a portion of a trace up to a certain boundary.

After. It identifies a portion of a trace starting from a certain boundary.

Between-And. It identifies portion(s) of a trace delimited by two boundaries.

After-Until. This scope is similar to *Between-and*, with the difference that each identified segment extends to the right in case the event defined by the second boundary does not occur.

The nine patterns are:

Absence. It describes a portion of a system's execution that is free of certain events or states, as in "it is never the case that X holds".

Universality. It describes a portion of a system's execution that contains only states that have a desired property, as in "it is always the case that X holds".

Existence. It describes a portion of a system's execution that contains an instance of certain events or states, as in "X eventually holds".

Bounded existence. It describes a portion of a system's execution that contains at most a specified number of instances of a designated state transition or event, as in "it is always the case that event X occurs at most 2 times".

Precedence. It describes relationships between a pair of events (or states), where the occurrence of the first is a necessary pre-condition for an occurrence of the second, as in "it is always the case that if X holds, then Y previously held".

Response. It describes cause-effect relationships between a pair of events (or states), where an occurrence of the first must be followed by an occurrence of the second, as in "it is always the case that if X holds, then Y eventually holds".

Response chains. It is a generalization of the response pattern, as it describes relationships between *sequences* of individual states (or events), as in "it is always the case that if W holds, and is succeeded by X, then Z eventually holds after Y".

Precedence chains. It is a generalization of the precedence pattern, as it describes relationships between *sequences* of individual states (or events), as in "it is always the case that if X holds, then Y previously held and was preceded by X".

Constrained chain patterns. It describes a variant of response and precedence chain patterns that restricts

[3] A detailed description is available at http://patterns.projects.cis.ksu.edu.

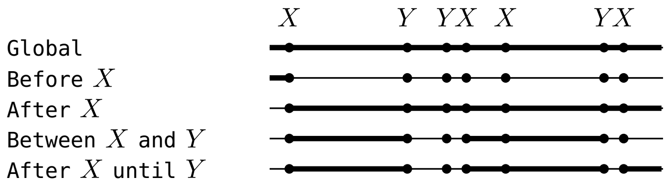


Fig. 1: Scopes in the catalogue of property specification patterns in [27]

user specified events from occurring between pairs of states (or events) in the chain sequences. This pattern has not been included in the definition of *TemPsy*.

Absence, Universality, Existence and Bounded Existence belong to the *Occurrence* category, while Precedence, Response, and Chains belong to the *Order* category.

3 The *TemPsy* language

As discussed in section 1, the ultimate goal of this work is to present a practical and scalable solution for the offline checking of the temporal requirements of a system with respect to a business process model, motivated by real and specific requirements in eGovernment systems. In this section we present the first step to achieve this goal, which is represented by the definition of the *TemPsy* language for the specification of temporal requirements of business processes, which will then be checked on an execution trace using the procedure described in section 6.

3.1 Eliciting the requirements of the language

The design of *TemPsy* has been driven by the analysis of the requirements of various applications developed as business processes by CTIE. We analyzed several applications and scrutinized the requirements specifications associated with all use cases and business process descriptions.

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. [27]. In some cases, we extended the original definitions proposed in [27] to match the specifications. For example, we extended the definitions of scopes to support references to a specific occurrence of an event (not only the first one as in [27]), as in the requirement "event A shall occur before the *second* occurrence of event X". Another variant of this type of scope boundary that we found is the one with requirements on the distance between events, such as "event A shall occur *five time units before the second* occurrence of event X". In other cases, the requirements specifications had to be expressed in terms of some real-time specification patterns [38, 43], which quantitatively define distance among events and durations of events.

3.2 Design

The analysis of the requirements specifications mentioned above made us ponder over the design of the specification language for expressing them.

The intrinsic temporal nature of the requirements specifications we found, including also constraints on the distance between events, could have suggested to follow the direction of building on some (metric) temporal logic. However, this decision would have not allowed us to fulfill requirement R1 (see section 1). One of the motivations behind this requirement is that specification languages based on temporal logic require a certain mathematical knowledge that is not common among practitioners.

Another design option would have been to consider the specification languages defined in the MDE community, namely temporal extensions of OCL, such as [18, 23, 33, 34, 42, 45, 46, 57, 59, 62]. However, these temporal extensions either include temporal logic operators—thus intrinsically inheriting the limitations of other specification approaches based on temporal logic, and not fulfilling requirement R1—or are pattern-based but have limited expressiveness. For example, none of the pattern-based OCL temporal extensions can express a property like "If the physical information of the card requester is collected within three days after the second approval notification, the card will be produced and then issued to the requester", which contains a reference to a specific occurrence of an event in a scope boundary, as well as an explicit temporal distance from the scope boundary event.

Based on the discussions with business analysts, and keeping in mind the goal of fulfilling requirement R1 above, we decided that *TemPsy* should have the following features: be as close to natural language as possible, make no use of mathematical constructs, and support the commonly-understood concepts (i.e., property specification patterns) used in the specification of requirements in the domain of business process modeling.

We designed *TemPsy* as a language sporting a syntax close to natural language, with all the constructs required to express the property specification patterns found in the business process applications developed by our partner, and a precise semantics expressed in terms of linear temporal traces. *TemPsy* supports all the patterns and scopes defined in [27], with the following extensions:

- 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.

- 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...".

- Support for expressing time distance between events occurrences in the precedence and response patterns as well as in their chain versions, for expressing properties

⟨*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])*

Fig. 2: Syntax of *TemPsy*

such as “event B should occur in response to event A within 2 time units”.

- Additional variants for the bounded existence and absence patterns.

3.3 Syntax

The syntax of *TemPsy* is shown in Fig. 2: 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 ⟨*TemPsyBlock*⟩ comprises a set of conjuncted ⟨*TemPsyExpression*⟩s. Each *TemPsy* expression starts with an optional ‘`temporal`’ keyword plus an alphanumeric identifier, followed by a ⟨*Scope*⟩ and a ⟨*Pattern*⟩. The keywords indicating the five ⟨*Scope*⟩s identify univocally the corresponding scopes from [27] (see section 2). As for the ⟨*Pattern*⟩s, ‘`always`’ corresponds to universality, ‘`eventually`’ to existence, ‘`never`’ to absence, ‘`preceding`’ to precedence and precedence chain, ‘`responding`’ to response and response chain.

The definitions of ⟨*Scope*⟩s and ⟨*Pattern*⟩s refer to the concept of ⟨*Event*⟩. We assume that an ⟨*Event*⟩ is represented by an alphanumeric string, to match the event strings 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).

3.4 *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 execution trace shown in Fig. 3 and for each property[4] indicate whether it is violated or not by the trace. First, we define the properties in English:

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

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

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)

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

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)

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

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)

The corresponding *TemPsy* expressions are shown below:

- `temporal p1: after 2` X `exactly 8 tu eventually` C
- `temporal p2: after` X `at most 30 tu eventually` A
- `temporal p3: after 1` X `at least 3 tu until` Y `eventually` C

[4] 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.

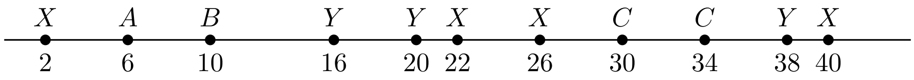


Fig. 3: An event trace on which to evaluate the properties described in section 3.4; events are above the line, timestamps below

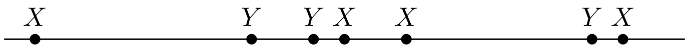


Fig. 4: A sample trace for the description of scopes

- temporal p4: after 2 X eventually exactly 2 C
- temporal p5: between X at least 5 tu and Y eventually at least 1 C
- temporal p6: before Y at least 3 tu eventually B
- temporal p7: before Y A, B responding at least 3 tu X

3.5 Informal Semantics

In this section we present the informal semantics of the scopes and the patterns supported in *TemPsy* expressions; they correspond to non-terminals $\langle Scope \rangle$ and $\langle Pattern \rangle$, respectively. In the following, symbols A, B, C, D, X, Y, Z represent strings that can be derived from non-terminal $\langle Event \rangle$; 'm', 'm1', 'm2', 'n', 'n1', and 'n2' are integers derived from the non-terminal $\langle Int \rangle$; 'tu' stands for "time unit(s)". The complete definition of the formal semantics of *TemPsy* can be found in section 5.

3.5.1 Scopes

For the description of scopes, we refer to the trace of events depicted in Fig. 4; to avoid cluttering, the figure does not show the events not used in the explanations. We use symbols X and Y as shorthands for events that can be derived from the non-terminal $\langle Event \rangle$.

Globally. This scope corresponds to the entire trace shown in Fig. 4.

Before. The general template for this scope in *TemPsy* is "before [m] X [$\langle ComparingOp \rangle$ n tu]"; it can be expanded in four forms: 1) "before X", 2) "before X $\langle ComparingOp \rangle$ n tu", 3) "before m X", 4) "before m X $\langle ComparingOp \rangle$ n tu". The first two forms are convenient shorthands for the third and fourth ones, respectively, with m $=1$. The form "before m X" selects the portion of the trace up to the m-th occurrence of event X; see, for example, the top row in Fig. 5a, where the interval from the origin of the trace up to the third occurrence of X is highlighted with a thick line. The form "before m X $\langle ComparingOp \rangle$ n tu" has three variants, depending on the possible expansions of non-terminal $\langle ComparingOp \rangle$:

- "before m X at least n tu" identifies the scope from the origin of the trace up to n time units before the m-th occurrence of X;
- "before m X at most n tu" identifies the scope starting at n time units before the m-th occurrence of X and bounded to the right by the m-th occurrence of X;
- "before m X exactly n tu" pinpoints the time instant at n time units before the m-th occurrence of X.

Examples of the first two variants of scopes are shown with thick segments in the second and third rows of Fig. 5a; for the last variant, see the last row of Fig. 5a, where the time instant selected by the scope is enclosed with a circle. In all examples, we have m$=3$ and n$=2$.

After. It has a dual semantics with respect to the *before* scope. We provide an intuition of its semantics using Fig. 5b.

Between-And. The general template for this scope in *TemPsy* is "between [m1] X [at least n1 tu] and [m2] Y [at least n2 tu]"; it can be expanded in four forms:

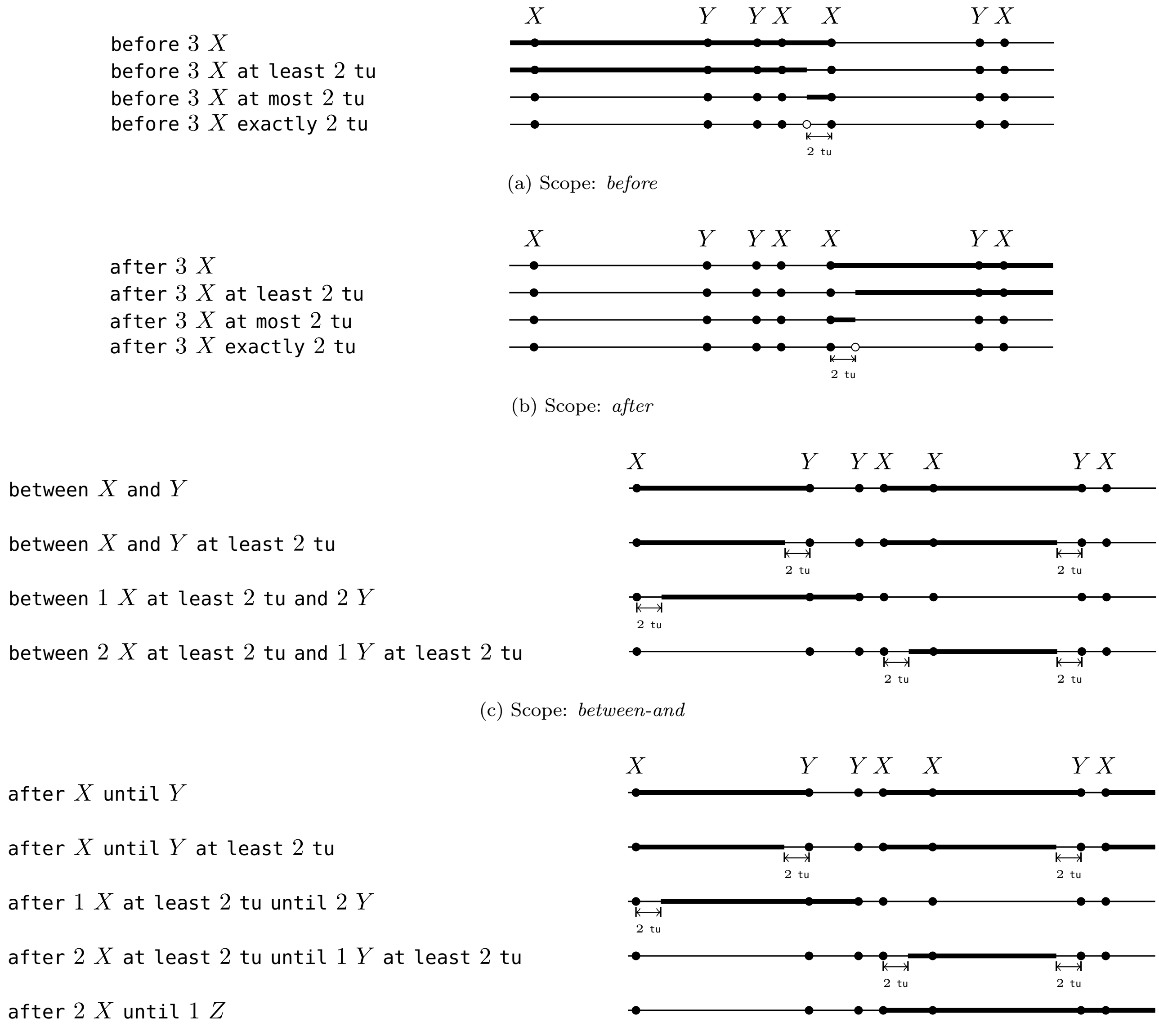
- "between m_1 X [at least n_1 tu] and m_2 Y [at least n_2 tu]";
- "between X [at least n_1 tu] and m_2 Y [at least n_2 tu]";
- "between m_1 X [at least n_1 tu] and Y [at least n_2 tu]";
- "between X [at least n_1 tu] and Y [at least n_2 tu]".

The first form is the most general: it selects the single segment of the trace delimited by the m_1-th occurrence of event X and the m_2-th occurrence of event Y happening after the m_1-th occurrence of X. The second and third forms are shorthands for the first one, with m1$=1$ and m2$=1$, respectively. The fourth form is the closest to the original definition in [27], since it selects all the segments in the trace delimited by the boundaries. In this regard, notice the difference with respect to the expression "between 1 X and 1 Y", which selects the segment delimited by the first occurrence of X and the first occurrence of Y after X. In all forms it is possible to use the expression at least n tu when defining boundaries, with the same meaning described for the scope *before*. Four examples of the *Between-and* scope are shown in Fig. 5c.

After-Until. This scope is similar to *Between-and*, with the difference that each identified segment extends to the right in case the event defined by the second boundary does not occur; this peculiarity can be noticed in the first two rows of Fig. 5d (also by comparing them with the corresponding ones in Fig. 5c), as well as in the last row.

Note that all scopes are open on the bounds delimited by the boundary events themselves, i.e., in general[5], the *before* scope is closed on the left bound and open on the right bound; the *after* scope is open on the left bound, and closed on the right bound; the *between-and* scope is open on both bounds; the *after-until* scope is open on both bounds when the right boundary event occurs, or is open on the left and closed on the right when the right boundary event does not occur.

[5] The scopes that contain constraints on time distance from the boundary events (with "at least" and "exactly") are closed on the bounds



(a) Scope: *before*

(b) Scope: *after*

(c) Scope: *between-and*

(d) Scope: *after-until*

Fig. 5: Examples of *TemPsy* scopes

3.5.2 Patterns

TemPsy supports eight of the nine patterns defined in [27]. Their semantics has been already briefly explained in section 2; below we only highlight the semantics for the patterns that have been extended upon inclusion in *TemPsy*.

Existence. This pattern comes in four forms:

- "`eventually` A" indicates that event A will eventually happen at least once;
- "`eventually at least m` A" indicates that event A will eventually happen at least m times;
- "`eventually at most m` A" indicates that event A will eventually happen at most m times;.
- "`eventually exactly m` A" indicates that event A will eventually happen exactly m times.

The last three forms are variants of the *bounded existence* pattern, a subclass [2] of the *existence* one.

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

Precedence. This pattern (also available in the variant called *precedence chain*) indicates the precondition relationship between a pair of events (respectively, the two blocks of a chain) in which the occurrence of the second event (respectively, block) depends on the occurrence of the first event (respectively, block). Based on this original definition, we added support for timing information to enable expressing the time distance between two adjacent events. The semantics can be explained using the following example and the event trace in Fig. 6; the expression "A `preceding at most 10 tu` B, $\#$`at least 5 tu` C" indicates that the event A is the precondition of the block "B followed by C", that the time distance between A and B should be at most 10 time units, and the time distance (expressed using the `#` symbol) between events B and C should be at least 5 time units. Here, A (left-hand side of '`preceding`') represents the first block of the chain, while the expression "B, $\#$`at least 5 tu` C" represents the second block (right-hand side of '`preceding`').

Response. This pattern (also available in the variant called *response chain*) specifies the cause-effect relationship between a pair of events (respectively, the two blocks of a chain) in which the occurrence of the first event (re-

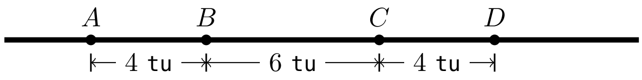


Fig. 6: Example trace for illustrating the precedence and response patterns

spectively, first block) leads to the occurrence of the second event (respectively, second block). Similarly to the previous pattern, we added support for timing information to enable expressing the time distance between two adjacent events. The semantics can be explained using the following example and the event trace in Fig. 6; the expression "C, D `responding at most 10 tu` A, `#at least 5 tu` B" specifies that two successive events A and B stimulate the sequential occurrence of C and D, the time interval between A and B should be at least 5 time units, and the time interval between B (second element of the first block) and C (first element of the second block) should be at most 10 time units. This property is violated by the example in Fig. 6, because the time distance between A and B is only 4 time units.

3.6 Expressivity

As discussed earlier, the main goal of *TemPsy* is to make as easy as possible the specification of the temporal requirements of business processes, by supporting—in an intuitive way—only the constructs needed to express temporal requirements commonly found in business process applications. Hence, by design, *TemPsy* does not aim at being as expressive as a full-fledged temporal logic.

More precisely, *TemPsy* can specify *only* the expressions resulting from the combination of one of the five supported scopes (and their variants) with one of the eight supported patterns (and their variants). For each of these expressions, it is possible to write a formula with the same meaning in a full-fledged temporal logic like MTL [44] (see, for example, the syntax-directed translation of property specification patterns, targeting MTL, proposed in [2]). On the other hand, all the MTL formulae that do not correspond to one of the ⟨scope, pattern⟩ combinations *cannot* be expressed in *TemPsy*.

In our context, this limitation turns out to be more theoretical than practical, since we were able to express in *TemPsy* all the requirements of the business processes of our case study. Nevertheless, as part of future work, we plan to assess the expressivity of *TemPsy* by applying it for the specification of business processes in other application domains.

4 Applying *TemPsy* in an eGovernment scenario

In this section we report on the application of *TemPsy* for the specification of a business process extracted from the case study developed with our partner. After illustrating the conceptual and behavioral models of some fragments of the business process application, we present some requirements specifications associated with these business process fragments and show how these specifications can be expressed in *TemPsy*. We also discuss the adoption and use of *TemPsy* by our partner.

Notice that the case study description has been sanitized, for the purpose of not disclosing confidential information, and simplified, to obtain a model at the minimum level of detail required to illustrate and express the requirement specifications.

4.1 Business process models

We consider the *Identity Card Management (ICM)* business process, which is in charge of issuing and managing the ID cards of the diplomatic personnel of the country. Its conceptual model is shown in Fig. 7, while three activity diagrams corresponding to process fragments are sketched in Fig. 8.

The conceptual model includes the `ICM` class, which manages `Cards` and `Requests` (for new cards). The `ICM` class has methods that deal with approval/rejection of card requests, card production and issuance, and card loss/expiration. Class `Card` has methods to query about the state of the card, which can be lost, found, expired, or returned (to the administration).

The activity diagram in Fig. 8a shows the business process fragment for processing a card request. Once a request for a card is submitted to the *ICM* system, it is evaluated and then either approved or rejected. Afterwards, a notification letter of approval or rejection is sent to the requester. Upon approval, the requester is asked to provide her physical information (e.g., hair and eye color, height) to the *ICM* system. In case this information is not provided, a second notification is sent; if the requester does not show up after two notifications, the request is then rejected and the requester notified about it. If the requester provides her information, the *ICM* system requests the production of the physical card, which is then issued to the requester.

The business process fragment executed in case of card loss is depicted in Fig. 8b. The *ICM* system first registers the card loss case and issues a temporary card to the card holder. If the lost card is found before the production of a new one, the *ICM* system recalls the temporary card. After the production of a new card, the *ICM* system will recall the temporary card and issue the new one. If the lost card is found after the production of the new one but before the recall of the temporary one, the *ICM* system will recall the old card before recalling the temporary one.

The activity diagram in Fig. 8c corresponds to the business process fragment executed in case of card expiration. When a card expires, the *ICM* system sends the card holder a letter to recall the card. If the card is returned, a confirmation receipt is then sent to the card holder; otherwise, another recall letter is sent to her. If, after two notification letters, the card holder has not returned the card yet, the *ICM* system reports the case to the police and the card holder will be fined.

4.2 Requirement specifications

We now list some requirements specifications associated with the three fragments of the *ICM* business process, and show how they can be expressed in *TemPsy*. These nine specifications (three for each business process fragment) have been selected out of the 47 available for the *ICM* application. Notice that these specifications have been written by the business analysts of our partner, who have do-

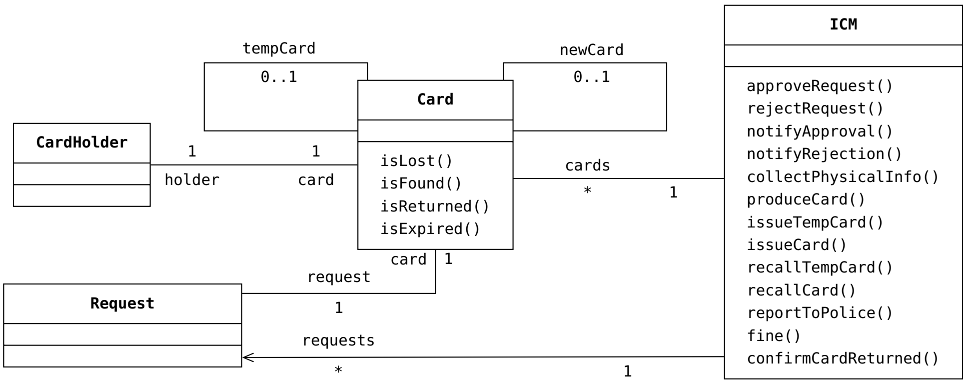
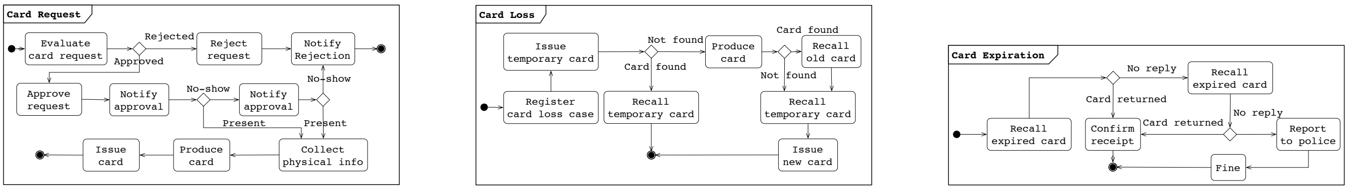


Fig. 7: Conceptual model of the ICM business process



(a) *card request* fragment (b) *card loss* fragment (c) *card expiration* fragment

Fig. 8: Activity diagrams of three fragments of the ICM business process

main knowledge, and represent realistic properties being used in practice.

Card Request:

R1 Once a card request is approved, the requester is notified within three days; this notification has to occur before the production of the card is started.

R2 The requester has to show up for the collection of her physical information within five days from the first notification.

R3 If the physical information of the requester is collected within three days after the second approval notification, the card will be produced and then issued to the requester.

These requirements specifications can be expressed in *TemPsy* as follows:

```
1  temporal R1:
2    before ICM.issueCard
3    ICM.notifyApproval
4    responding at most 3*24*3600 tu
5    ICM.approveRequest
6  temporal R2:
7    after 1 ICM.notifyApproval
8    at most 5*24*3600 tu
9    eventually ICM.collectPhysicalInfo
10 temporal R3:
11   after 2 ICM.notifyApproval
12   at most 3*24*3600 tu
13   ICM.collectPhysicalInfo
14   preceding
15   ICM.produceCard, ICM.issueCard
```

Property R1 is expressed in lines 1–5. The *before* scope is delimited by the event `ICM.issueCard`. The *response* pattern is bounded (time units are expressed in seconds) and requires the notification to the requester (`ICM.notifyApproval`) to happen in response to the action of approving the request (`ICM.approveRequest`). Property R2 (lines 6–9) combines an *after* scope with an *existence* pattern. In R3, the *after* scope (line 11) is bounded by the second occurrence of `ICM.notifyApproval`; this scope is associated with a *precedence chain* pattern, where `ICM.collectPhysicalInfo` represents the first block and the events chain `ICM.produceCard`, `ICM.issueCard`, the second block.

Card Loss:

L1 If a card is reported as lost, a temporary card will be issued to the card holder within one day, and will be recalled in ten days after the issuance.

L2 After a card has been registered as lost, a new card should be produced at least two days before its issuance.

L3 If the lost card is found after the production of a new card, the old card and the temporary one should be recalled within three days.

These requirements specifications can be expressed in *TemPsy* as follows:

```
temporal L1:
  after Card.isLost
  at most 24*3600 tu
  ICM.recallTempCard
  responding at most 10*24*3600 tu
  ICM.issueTempCard
temporal L2:
  after Card.isLost
  ICM.produceCard
  preceding at least 2*24*3600 tu
  ICM.issueCard
temporal L3:
  after Card.isLost
  until ICM.issueCard
```

```
  ICM.recallCard,
  ICM.recallTempCard,
  responding at most 3*24*3600 tu
  ICM.produceCard,
  Card.isFound
```

Property L1 contains an *after* scope and a *response* pattern, where the scope boundary contains a time constraint, and the pattern also restricts the time distance between the issuance of a temporary card (`ICM.issueTempCard`) and the corresponding card recall event (`ICM.recallTempCard`). Property L3 combines an *after-until* scope with a *precedence chain* pattern, where the first block corresponds to the events chain `ICM.recallCard`, `ICM.recallTempCard`, and the second block corresponds to the events chain `ICM.produceCard`, `Card.isFound`.

Card Expiration:

E1 Once a card expires, the holder is notified to return the card at most twice.

E2 In case the expired card has not been returned after five days from the second notification to the holder, the latter will be fined after the case will be reported to the police.

E3 Once a card is returned, the holder will receive a confirmation within one day.

These requirements specifications can be expressed in *TemPsy* as follows:

```
temporal E1:
  after Card.isExpired
  until Card.isReturned
  eventually at most 2 ICM.recallCard
temporal E2:
  after 2 ICM.recallCard
  at least 5*24*3600 tu
  until Card.isReturned
  ICM.fine
  responding
  ICm.reportToPolice
temporal E3:
  globally
  ICM.confirmCardReturned
  responding at most 24*3600 tu
  Card.isReturned
```

Property E1 uses an *after-until* scope, where the left boundary event corresponds to the expiration of the card (`Card.isExpired`) and the right boundary event corresponds to the return of the card (`Card.isReturned`). A *bounded existence* pattern is used to specify the maximum amount of notifications (`ICM.recallCard`) that can occur. In property E2 we use an *after-until* scope combined with the keyword '`at least`' for the first boundary, to delimit the period during which the card holder will be fined once the expiration case is reported to the police (`ICM.reportToPolice`). Property E3 states an invariant of the system (using the *globally* scope) for the *response* pattern that correlates the return of the card (`Card.isReturned`) to the confirmation to the holder (`ICM.confirmCardReturned`).

4.3 Adoption of *TemPsy* by our partner

Our partner has adopted *TemPsy* as the specification language for expressing the requirements of its business process models. *TemPsy* specifications have provided business analysts with a means to reason and formalize business process requirements, and have replaced informal specifications written in natural language. Our partner has also developed, for internal use, a graphical version of *TemPsy*, which has been integrated into the SoftwareAG ARIS modeling tool [60], as part of the Prometa business process modeling framework[6]; although the illustration of the graphical notation for *TemPsy* is out of the scope of this paper, we provide an example of it in Fig. 9.

In terms of expressiveness, we recall that *TemPsy* has been designed based on the analysis of the structure of the requirements specifications written by our partner. Hence, all the requirements of the case study presented in the previous section could be expressed with *TemPsy*. Table 1 shows the distribution of the 47 requirements of the *ICM* business process, in terms of the combination of scopes and patterns.

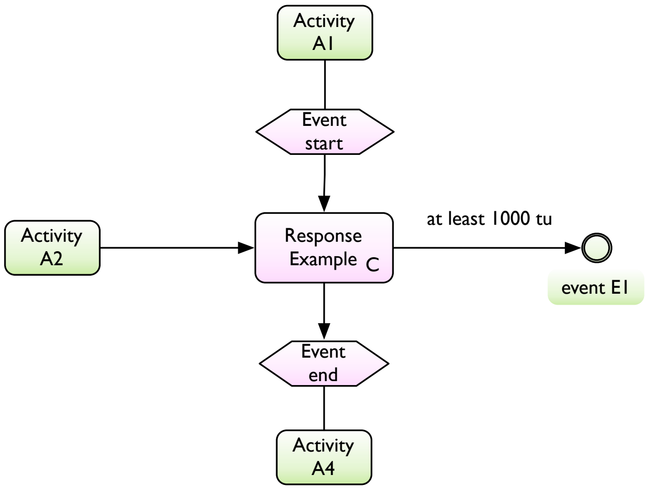


Fig. 9: Example of the graphical notation for *TemPsy*

Table 1: Distribution of requirements from the *ICM* business process in terms of the combination of scopes and patterns

scope+pattern	# of requirements
globally+universality	1
globally+absence	1
globally+existence	2
globally+precedence	4
globally+response	4
before+absence	1
before+existence	2
before+precedence	3
before+response	2
after+universality	1
after+absence	1
after+existence	4
after+precedence	3
after+response	2
between-and+universality	2
between-and+absence	2
between-and+existence	1
between-and+precedence	1
between-and+response	1
after-until+universality	1
after-until+absence	1
after-until+existence	4
after-until+precedence	1
after-until+response	2

[6] `https://joinup.ec.europa.eu/community/nifo/case/prometa-organisational-interoperability-framework-eservice-design-luxemburg`.

5 Formal Semantics of *TemPsy*

This section presents the formal semantics of *TemPsy*, using the concept of temporal linear traces.

5.1 Events and Trace

Event. An atomic event e is an element of the set Σ, which contains all the symbolic strings corresponding to operations recorded in a trace or log.

EventChain. An *EventChain* is a chain of *Events* occurring in sequence, with an optional quantification of the time distance between each pair of adjacent elements. An m-length EventChain ($m > 1$) is denoted as $e_1, t_1, e_2, \ldots, t_{m-1}, e_m$. The symbol t_i (with $1 \leq i \leq m-1$) represents the time distance between e_i and e_{i+1} (if defined) and has the form $t_i = \# \bowtie_i \delta_i$ `tu` with $\delta_i \in \mathbb{N}^+$ and $\bowtie_i \in \{$`at least, at most, exactly`$\}$; when t_i is undefined we use the notation $t_i = \perp$. Function $len(EC)$ returns the length m of an m-length EventChain EC.

Trace. A n-length *trace* λ is a finite sequence of atomic events $(e_0, \ldots, e_{n-1})$, where e_0 is its starting event and n is the length. The universal set of sub-traces is denoted as Λ.

We assume that each event in a trace is timestamped and that there is a function $\tau : \mathbb{N} \to \mathbb{N}$, which returns the timestamp $\tau(i)$ at which the event in position i of the trace occurred. The timestamp is a natural number and represents the absolute value of time with respect to the time unit defined for the system. Given a trace λ we assume that the sequence of timestamps $\tau(0), \tau(1), \ldots, \tau(n-1)$ is strictly monotonic, i.e., $\tau(i) < \tau(i+1)$ for all i, with $0 \leq i \leq n-2$.

We now introduce some notations used in the rest of the section. Given an n-length trace λ,

- $\lambda(i)$ denotes the atomic event at position i in the trace, with $0 \leq i \leq n-1$;
- $td(i, j)$ denotes the time distance between $\lambda(i)$ and $\lambda(j)$ and is defined as $td(i, j) \equiv \tau(j) - \tau(i)$, with $0 \leq i \leq j \leq n-1$;
- $\lambda(i : j)$ denotes the sub-trace of λ from $\lambda(i)$ to $\lambda(j)$ including both bounds, with $0 \leq i \leq j \leq n-1$;
- $\#(\lambda, i, j, e)$ denotes the number of occurrences of event e in the sub-trace $\lambda(i : j)$ of λ.

5.2 Temporal Expressions

In the following definitions, let e, e_1, e_2 be atomic events; EC_1, EC_2 be event chains; n be the length of a trace; b, d be positive natural numbers denoting time distances; a, c denote the specific occurrence of a scope boundary event and range over $\{0, \ldots, n-1\}$ if defined or be equal to $\{\perp\}$ if undefined; $\alpha, \alpha', \beta', \gamma, \theta, \theta', \eta, \eta'$ be auxiliary variables ranging over $\{0, \ldots, n-1\}$.

Scopes. Let $\mathbb{S}$ be the set of scopes that can be derived from the non-terminal $\langle Scope \rangle$ in the grammar in Fig. 2. A scope $s \in \mathbb{S}$ is a set of sub-traces of an n-length trace $\lambda \in \Lambda$ defined by the function $\phi_{[s]}(\lambda) : \Lambda \to 2^{\Lambda}$ as follows:

globally: $\phi_{[\texttt{globally}]}(\lambda) = \{\lambda\}$

before:

- $\phi_{[\texttt{before}\ a\ e]}(\lambda) = \{\lambda(0 : \theta - 1) \mid \theta \geq 1, \lambda(\theta) = e, \#(\lambda, 0, \theta, e) = m\}$
- $\phi_{[\texttt{before}\ a\ e\ \texttt{at least}\ b\ \texttt{tu}]}(\lambda) = \{\lambda(0 : \theta') \mid \lambda(\theta) = e, \theta' = \max(\{\gamma \mid td(\gamma, \theta) \geq b\}), \#(\lambda, 0, \theta, e) = m\}$
- $\phi_{[\texttt{before}\ a\ e\ \texttt{at most}\ b\ \texttt{tu}]}(\lambda) = \{\lambda(\theta' : \theta - 1) \mid \lambda(\theta) = e, \theta' = \max(\{\gamma \mid td(\gamma, \theta) \geq b\}), \#(\lambda, 0, \theta, e) = m\}$
- $\phi_{[\texttt{before}\ a\ e\ \texttt{exactly}\ b\ \texttt{tu}]}(\lambda) = \{\lambda(\theta' : \theta') \mid \lambda(\theta) = e, \theta' = \max(\{\gamma \mid td(\gamma, \theta) \geq b\}), \#(\lambda, 0, \theta, e) = m\}$

where $m = \begin{cases} 1, & \text{if } a = \perp \\ a, & \text{else} \end{cases}$

after:

- $\phi_{[\texttt{after}\ a\ e]}(\lambda) = \{\lambda(\theta + 1 : n - 1) \mid \theta \leq n - 2, \lambda(\theta) = e, \#(\lambda, 0, \theta, e) = m\}$
- $\phi_{[\texttt{after}\ a\ e\ \texttt{at least}\ b\ \texttt{tu}]}(\lambda) = \{\lambda(\theta' : n - 1) \mid \lambda(\theta) = e, \theta' = \min(\{\gamma \mid td(\theta, \gamma) \geq b\}), \#(\lambda, 0, \theta, e) = m\}$
- $\phi_{[\texttt{after}\ a\ e\ \texttt{at most}\ b\ \texttt{tu}]}(\lambda) = \{\lambda(\theta + 1 : \theta') \mid \lambda(\theta) = e, \theta' = \min(\{\gamma \mid td(\theta, \gamma) \geq b\}), \#(\lambda, 0, \theta, e) = m\}$
- $\phi_{[\texttt{after}\ a\ e\ \texttt{exactly}\ b\ \texttt{tu}]}(\lambda) = \{\lambda(\theta' : \theta') \mid \lambda(\theta) = e, \theta' = \min(\{\gamma \mid td(\theta, \gamma) \geq b\}), \#(\lambda, 0, \theta, e) = m\}$

where $m = \begin{cases} 1, & \text{if } a = \perp \\ a, & \text{else} \end{cases}$

between-and:

- $\phi_{[\texttt{between}\ e_1\ \texttt{and}\ e_2]}(\lambda) = \{\lambda(\alpha_k + 1 : \beta_k - 1) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1\}$
- $\phi_{[\texttt{between}\ e_1\ \texttt{and}\ e_2\ \texttt{at least}\ d\ \texttt{tu}]}(\lambda) = \{\lambda(\alpha_k + 1 : \beta'_k) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \beta'_k = \max(\{\gamma \mid td(\gamma, \beta_k) \geq d\})\}$
- $\phi_{[\texttt{between}\ e_1\ \texttt{at least}\ b\ \texttt{tu and}\ e_2]}(\lambda) = \{\lambda(\alpha'_k : \beta_k - 1) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \alpha'_k = \min(\{\gamma \mid td(\alpha_k, \gamma) \geq b\})\}$
- $\phi_{[\texttt{between}\ e_1\ \texttt{at least}\ b\ \texttt{tu and}\ e_2\ \texttt{at least}\ d\ \texttt{tu}]}(\lambda) = \{\lambda(\alpha'_k : \beta'_k) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \alpha'_k = \min(\{\gamma \mid td(\alpha_k, \gamma) \geq b\}), \beta'_k = \max(\{\gamma \mid td(\gamma, \beta_k) \geq d\})\}$
- $\phi_{[\texttt{between}\ a\ e_1\ \texttt{and}\ c\ e_2]}(\lambda) = \{\lambda(\alpha + 1 : \beta - 1) \mid \lambda(\alpha) = e_1, \#(\lambda, 0, \alpha, e_1) = x, \lambda(\beta) = e_2, \#(\lambda, \alpha + 1, \beta, e_2) = y\}$
- $\phi_{[\texttt{between}\ a\ e_1\ \texttt{and}\ c\ e_2\ \texttt{at least}\ d\ \texttt{tu}]}(\lambda) = \{\lambda(\alpha + 1 : \beta') \mid \lambda(\alpha) = e_1, \#(\lambda, 0, \alpha, e_1) = x, \lambda(\beta) = e_2, \#(\lambda, \alpha + 1, \beta, e_2) = y, \beta' = \max(\{\gamma \mid td(\gamma, \beta) \geq d\})\}$
- $\phi_{[\texttt{between}\ a\ e_1\ \texttt{at least}\ b\ \texttt{tu and}\ c\ e_2]}(\lambda) = \{\lambda(\alpha' : \beta - 1) \mid \lambda(\alpha) = e_1, \#(\lambda, 0, \alpha, e_1) = x, \lambda(\beta) = e_2, \#(\lambda, \alpha + 1, \beta, e_2) = y, \alpha' = \min(\{\gamma \mid td(\alpha, \gamma) \geq b\})\}$

- $\phi_{[\texttt{between } a\ e_1 \texttt{ at least } b \texttt{ tu and } c\ e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) = \{\lambda(\alpha' : \beta') \mid \lambda(\alpha) = e_1, \#(\lambda, 0, \alpha, e_1) = x, \lambda(\beta) = e_2, \#(\lambda, \alpha+1, \beta, e_2) = y, \alpha' = \min(\{\gamma \mid td(\alpha, \gamma) \geq b\}, \beta' = \max(\{\gamma \mid td(\gamma, \beta) \geq d\}\}$

where $x = \begin{cases} 1, & \text{if } a = \perp \\ a, & \text{else} \end{cases}$ and $y = \begin{cases} 1, & \text{if } c = \perp \\ c, & \text{else} . \end{cases}$

after-until:

- $\phi_{[\texttt{after } e_1 \texttt{ until } e_2]}(\lambda) = \phi_{[\texttt{between } e_1 \texttt{ and } e_2]}(\lambda) \cup \{\lambda(\eta+1 : n-1) \mid \eta = \min(\{\gamma \mid \gamma \leq n-2, \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n-1, \lambda(k) \neq e_2\})\}$
- $\phi_{[\texttt{after } e_1 \texttt{ until } e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) = \phi_{[\texttt{between } e_1 \texttt{ and } e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \cup \{\lambda(\eta + 1 : n-1) \mid \eta = \min(\{\gamma \mid \gamma \leq n-2, \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n-1, \lambda(k) \neq e_2\})\}$
- $\phi_{[\texttt{after } e_1 \texttt{ at least } b \texttt{ tu until } e_2]}(\lambda) = \phi_{[\texttt{between } e_1 \texttt{ at least } b \texttt{ tu and } e_2]}(\lambda) \cup \{\lambda(\eta' : n-1) \mid \eta' = \min(\{\gamma \mid td(\eta, \gamma) \geq b\}), \eta = \min(\{\gamma \mid \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n-1, \lambda(k) \neq e_2\})\}$
- $\phi_{[\texttt{after } e_1 \texttt{ at least } b \texttt{ tu until } e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) = \phi_{[\texttt{between } e_1 \texttt{ at least } b \texttt{ tu and } e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \cup \{\lambda(\eta' : n-1) \mid \eta' = \min(\{\gamma \mid td(\eta, \gamma) \geq b\}), \eta = \min(\{\gamma \mid \lambda(\gamma) = e_1, \forall k, \gamma < k \leq n-1, \lambda(k) \neq e_2\})\}$
- $\phi_{[\texttt{after } a\ e_1 \texttt{ until } c\ e_2]}(\lambda) = \phi_{[\texttt{between } a\ e_1 \texttt{ and } c\ e_2]}(\lambda) \cup \{\lambda(\eta + 1 : n-1) \mid \eta \leq n-2, \lambda(\eta) = e_1, \#(\lambda, 0, \eta, e_1) = x, \#(\lambda, \eta+1, n-1, e_2) < y\}$
- $\phi_{[\texttt{after } a\ e_1 \texttt{ until } c\ e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) = \phi_{[\texttt{between } a\ e_1 \texttt{ and } c\ e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \cup \{\lambda(\eta + 1 : n-1) \mid \eta \leq n-2, \lambda(\eta) = e_1, \#(\lambda, 0, \eta, e_1) = x, \#(\lambda, \eta+1, n-1, e_2) < y\}$
- $\phi_{[\texttt{after } a\ e_1 \texttt{ at least } b \texttt{ tu until } c\ e_2]}(\lambda) = \phi_{[\texttt{between } a\ e_1 \texttt{ at least } b \texttt{ tu and } c\ e_2]}(\lambda) \cup \{\lambda(\eta' : n-1) \mid \eta' = \min(\{\gamma \mid td(\eta, \gamma) \geq b\}), \lambda(\eta) = e_1, \#(\lambda, 0, \eta, e_1) = x, \#(\lambda, \eta+1, n-1, e_2) < y\}$
- $\phi_{[\texttt{after } a\ e_1 \texttt{ at least } b \texttt{ tu until } c\ e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) = \phi_{[\texttt{between } a\ e_1 \texttt{ at least } b \texttt{ tu and } c\ e_2 \texttt{ at least } d \texttt{ tu}]}(\lambda) \cup \{\lambda(\eta' : n-1) \mid \eta' = \min(\{\gamma \mid td(\eta, \gamma) \geq b\}), \lambda(\eta) = e_1, \#(\lambda, 0, \eta, e_1) = x, \#(\lambda, \eta+1, n-1, e_2) < y\}$

where $x = \begin{cases} 1, & \text{if } a = \perp \\ a, & \text{else} \end{cases}$ and $y = \begin{cases} 1, & \text{if } c = \perp \\ c, & \text{else} . \end{cases}$

Event and EventChain matching function. Let λ be an n-length trace, EC be an m-length EventChain ($1 \leq m \leq n$). The matching function *match* returns true if there is an occurrence of an event (or of an EventChain) in a certain position of the trace. For a 1-length EventChain $EC = e$, i.e., a single event, we have $match(\lambda, EC, i) = true$, with $i, 0 \leq i \leq n-1$, if $\lambda(i) = e$. For an event chain $EC = e_1, t_1, e_2, \ldots, t_{m-1}, e_m$, we have $match(\lambda, EC, i) = true$, with $i, 0 \leq i \leq n-m$, if there exist $i_1, i_2, \ldots, i_m \in \{0, \ldots, n-1\}$, such that $i_1 = i$, $i_{k+1} = i_k + 1, 1 \leq k \leq m-1$, $\lambda(i_1) = e_1, \lambda(i_2) = e_2, \ldots, \lambda(i_m) = e_m$ and for all $j, 1 \leq j \leq m-1$, such that $t_j \neq \perp$, we have:

$$\begin{cases} td(i_j, i_{j+1}) \geq \delta_j \text{ if } \bowtie_j = \texttt{at least}; \\ td(i_j, i_{j+1}) \leq \delta_j \text{ if } \bowtie_j = \texttt{at most}; \\ td(i_j, i_{j+1}) = \delta_j \text{ if } \bowtie_j = \texttt{exactly}. \end{cases}$$

For an events chain $EC = e_1, t_1, e_2, \ldots, t_{m-1}, e_m$ we also define two auxiliary functions $\mathit{first}(\lambda, EC, i)$ and $last(\lambda, EC, i)$, which return, respectively, the timestamp of the first and the last event of EC when the chain is matched in position i of the trace λ.

Patterns. Let $\mathbb{P}$ be the set of patterns that can be derived from the non-terminal $\langle Pattern \rangle$ in the grammar in Fig. 2. The semantics of a pattern $p \in \mathbb{P}$ is given by the function $\psi_{[p]}(\lambda) : \Lambda \to \{true, false\}$ defined as follows:

universality: $\psi_{[\texttt{always } e]}(\lambda) \Leftrightarrow \forall i, 0 \leq i \leq n-1, \lambda(i) = e$

absence:

- $\psi_{[\texttt{never } e]}(\lambda) \Leftrightarrow \forall i, 0 \leq i \leq n-1, \lambda(i) \neq e$
- $\psi_{[\texttt{never exactly } m\ e]}(\lambda) \Leftrightarrow \#(\lambda, 0, n-1, e) \neq m$

existence:

- $\psi_{[\texttt{eventually } e]}(\lambda) \Leftrightarrow \exists i, 0 \leq i \leq n-1, \lambda(i) = e$
- $\psi_{[\texttt{eventually } \bowtie\ m\ E]}(\lambda) \Leftrightarrow \#(\lambda, 0, n-1, e) \triangle\ m$

  where $\triangle = \begin{cases} \geq, & \text{if } \bowtie = \texttt{at least}; \\ \leq, & \text{if } \bowtie = \texttt{at most}; \\ =, & \text{if } \bowtie = \texttt{exactly}. \end{cases}$

precedence:

- $\psi_{[EC_1 \texttt{ preceding } EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n-1, match(\lambda, EC_2, i) \Rightarrow \exists j, 0 \leq j \leq i - len(EC_1), match(\lambda, EC_1, j)$
- $\psi_{[EC_1 \texttt{ preceding } \bowtie\ b \texttt{ tu } EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n-1, match(\lambda, EC_2, i) \Rightarrow \exists j, 0 \leq j \leq i - len(EC_1), match(\lambda, EC_1, j)$ and $(\mathit{first}(\lambda, EC_2, i) - last(\lambda, EC_1, j)) \triangle\ b$

  where $\triangle = \begin{cases} \geq, & \text{if } \bowtie = \texttt{at least}; \\ \leq, & \text{if } \bowtie = \texttt{at most}; \\ =, & \text{if } \bowtie = \texttt{exactly}. \end{cases}$

response:

- $\psi_{[EC_1 \texttt{ responding } EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n-1, match(\lambda, EC_2, i) \Rightarrow \exists j, i + len(EC_2) \leq j \leq n-1, match(\lambda, EC_1, j)$
- $\psi_{[EC_1 \texttt{ responding } \bowtie\ b \texttt{ tu } EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n-1, match(\lambda, EC_2, i) \Rightarrow \exists j, i + len(EC_2) \leq j \leq n-1, match(\lambda, EC_1, j)$ and $(\mathit{first}(\lambda, EC_1, i) - last(\lambda, EC_2, j)) \triangle\ b$

  where $\triangle = \begin{cases} \geq, & \text{if } \bowtie = \texttt{at least}; \\ \leq, & \text{if } \bowtie = \texttt{at most}; \\ =, & \text{if } \bowtie = \texttt{exactly}. \end{cases}$

Temporal Expression. The semantics over a trace λ of a temporal expression derived from the non-terminal $\langle TemPsyExpression \rangle$ containing a scope $s \in \mathbb{S}$ and a pattern $p \in \mathbb{P}$, represented as a pair $\langle s, p \rangle$, is defined as: $\lambda \models \langle s, p \rangle \Leftrightarrow \forall \lambda' \in \phi_{[s]}(\lambda),\ \psi_{[p]}(\lambda')$.

6 Model-driven Trace Checking of *TemPsy* properties

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

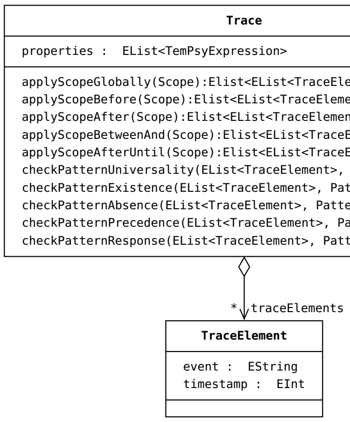


Fig. 10: Conceptual model for execution traces

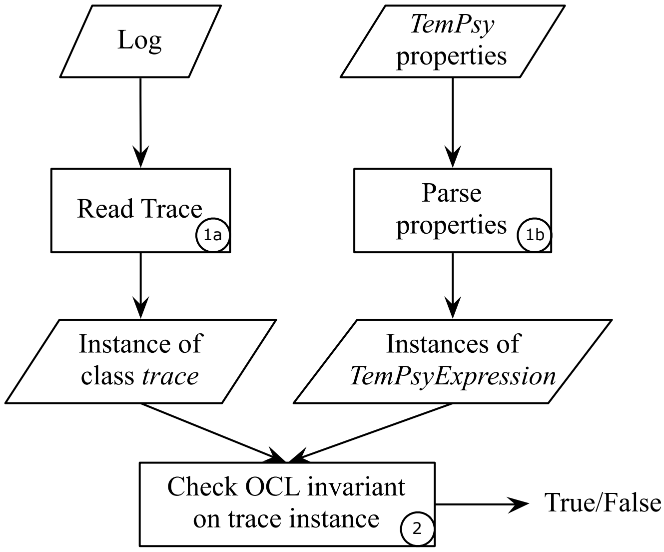


Fig. 11: Overview of the approach

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 [28], 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.

In the rest of this section, we first introduce the conceptual model we have defined to represent execution traces; afterwards, we provide an overview of our approach and show how *TemPsy* properties (decomposed in scopes and patterns) can be expressed as OCL constraints on the conceptual model. We conclude the section with an example of the application of the trace checking procedure and with some notes about the implementation of the approach in our TEMPSY-CHECK tool.

6.1 Conceptual model for execution traces

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

We propose a simple and yet generic model of system execution traces; it can be extended (by enriching the type of event) depending on the actual type of system (e.g., business process, access control framework) and the type of properties to check. The model, depicted in Fig. 10 with a UML class diagram, contains a `Trace`, which is composed of a sequence of `TraceElements`, accessed through the association `traceElements`. Each `TraceElement` contains an attribute `event` of type string, which represents the actual event recorded in the trace, and an attribute `timestamp` of type integer, which indicates the time at which the event occurred. Class `Trace` contains also an attribute `properties`, which is a collection of `TemPsyExpressions`[7], representing the properties to be checked on the trace.

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

6.2 Overview of the approach

Our approach for model-driven trace checking is sketched in Fig. 11: parallelogram shapes correspond to input/output artifacts, while rectangles correspond to steps in the approach. The two inputs are represented by a log, corresponding to the trace one wants to check, and by a set of *TemPsy* properties. The log file is read and converted (step 1a) to an instance of the class `trace` in the model shown in Fig. 10. The *TemPsy* properties are parsed and converted (step 1b) to instances of class `TemPsyExpression`.

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

We have defined this invariant on the `Trace` class, as shown in Fig. 12. For every *TemPsy* property provided in input (and referenced in the instance of the trace through the attribute `self.properties`, line 2), the invariant evaluates a boolean function, which conceptually corresponds to applying the semantics of the pattern used in the property (accessed through the expression `property.pattern`) on a set of sub-traces, as defined by the scope used in the property (accessed through the expression `property.scope`).

More specifically, the body of the invariant expression is a multi-way branch (defined through a sequence of `if` statements), which selects a certain branch based on the specific scope type used within the property. Within the body of a branch, first a function of the form `applyScope*S*` is called.

[7] Class `TemPsyExpression` belongs to the meta-model of the language (not shown here for space reasons) and represents objects corresponding to the non-terminal ⟨*TemPsyExpression*⟩ of the grammar shown in Fig. 2.

```
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::ScopeType::GLOBALLY then
5     let subtraces:Sequence(OrderedSet(TraceElement)) = applyScopeGlobally(scope) in
6     if pattern.type = TemPsy::PatternType::UNIVERSALITY then
7       subtraces->forAll(subtrace | checkPatternUniversality(subtrace, pattern))
8     else if pattern.type = TemPsy::PatternType::EXISTENCE then
9       subtraces->forAll(subtrace |  checkPatternExistence(subtrace, pattern))
10    else if pattern.type = TemPsy::PatternType::ABSENCE then
11      subtraces->forAll(subtrace | checkPatternAbsence(subtrace, pattern))
12    else if pattern.type = TemPsy::PatternType::PRECEDENCE then
13      subtraces->forAll(subtrace | checkPatternPrecedence(subtrace, pattern))
14    else if pattern.type = TemPsy::PatternType::RESPONSE then
15      subtraces->forAll(subtrace | checkPatternResponse(subtrace, pattern))
16    endif endif endif endif endif
17  else if scope.type = TemPsy::ScopeType::BEFORE then
18    ...
19  else if scope.type = TemPsy::ScopeType::AFTER then
20    ...
21  else if scope.type = TemPsy::ScopeType::BETWEENAND then
22    ...
23  else if scope.type = TemPsy:ScopeType::AFTERUNTIL then
24    ...
25  endif endif endif endif endif)
```

Fig. 12: OCL invariant for checking *TemPsy* properties on a trace

This function takes the scope used in the property as input and returns a collection of sub-traces, as defined by the scope semantics. Afterwards, the invariant invokes a function of the form `checkPattern*P*`, which checks whether the pattern used in the property holds on each sub-trace.

For instance, let us assume that the type of the scope of the *TemPsy* property provided in input is *globally* and that the type of the pattern used in the property is *response*. As shown in line 5, the function `applyScopeGlobally` is invoked to compute the sub-trace(s) defined by the `scope` parameter; the return value of this function is assigned to variable `subtraces`. The branch indicated on line 15 is then taken, which results in the evaluation of the boolean function `checkPatternResponse` on all the elements[8] of `subtraces`, to check whether the input parameter `pattern` holds on each sub-trace.

The complete OCL definition of the functions of the form `applyScope*S*` and `checkPattern*P*` is available in the appendix A. We illustrate examples of the `applyScope*S*` and `checkPattern*P*` operations in subsections 6.3 and 6.4, respectively; to ease legibility and conciseness, all the code snippets presented in these subsections are written using pseudocode.

6.3 OCL functions for scopes

In this section we illustrate two examples of the OCL functions that are used to apply a scope definition on a trace. We show the pseudocode of functions `applyScopeBefore` and `applyScopeBetweenAnd`, corresponding to the *before* and the *between-and* scopes. These functions take as input an object representing a scope in *TemPsy* and yield one or more segments of the trace (i.e., sub-trace(s)), as determined by the semantics of the scope.

[8] In the case of scope *globally*, only the variable `subtraces` will contain, by definition, only one trace.

6.3.1 Before

The definition of the function `applyScopeBefore` is shown in Algorithm 1. The input parameter *scope* is an instance of the *before* scope, and the output is a list that contains the trace segments as determined by the structure of *scope*. We assume the parameter *scope* to have the form "`before [m] X [op n tu]`" (see section 3.5), in which `op` stands for the comparison operator (i.e., "`at least`", "`at most`", or "`exactly`") used in the constraint that defines the time distance from the scope boundary event X.

The function starts by reading the parameters X, m, op, and n from the instance of the *before* scope (lines 1–4). In addition, we define and initialize to an empty list both variable *result* (to store the output value) and the auxiliary variable *segment* (for collecting intermediate trace elements). If the parameter m is omitted in the scope definition, variable m is replaced with the value 1 (line 6), according to the default semantics of the *before* scope. We then assign to variable t the timestamp of the m-th occurrence of event X in the trace (line 7). If t is defined, it means that the m-th occurrence of the event has been found in the trace. Lines 9–22 select a segment from the trace, based on the value of op. For example, when op is "`at least`", line 11 selects all the trace elements that occur at least n time unit(s) before the m-th occurrence of event X. If no time distance constraint is specified in the *scope* (line 20), the function selects the trace segment starting at the beginning of the trace and ending at the m-th occurrence of event X. The function ends by adding the *segment* selected from the trace to the output variable *result*.

6.3.2 Between-and

Algorithm 2 presents the definition of the function `applyScopeBetweenAnd`. This function takes as input an object representing an instance of the *between-and* scope and

Algorithm 1: applyScopeBefore

```
Input: scope : an instance of the before scope
       structured as "before [m] X [op n tu]"
Output: result : a list containing the trace segment
        as determined by the parameters of scope
X ← event name of the scope boundary
m ← index of the specific occurrence of event X
op ← comparison operator of the constraint on time
  distance
n ← time distance from the m-th occurrence of X
result ← [], segment ← []
if m = null then m ← 1
t ← timestamp of the m-th occurrence of event X
if t ≠ null then
    switch op do
        case "at least" do
            segment ← trace elements with timestamp
              t' satisfying t' ≤ t − n
        end
        case "at most" do
            segment ← trace elements with timestamp
              t' satisfying t − n ≤ t' < t
        end
        case "exactly" do
            segment ← trace elements with timestamp
              equal to t − n
        end
        otherwise do
            segment ← trace elements with timestamp
              t' satisfying t' < t
        end
    end
end
result.append(segment)
return result
```

Algorithm 2: applyScopeBetweenAnd

```
   Input: scope : an instance of the between-and scope
          structured as "between [m1] X [at least
          n1 tu] and [m2] Y [at least n2 tu]"
   Output: result : a list of trace segments, as
           determined by the parameters of scope
1  X ← event name of the left boundary
2  Y ← event name of the right boundary
3  m1 ← index of the specific occurrence of event X
4  m2 ← index of the specific occurrence of event Y
5  n1 ← lower bound of the time distance from the
     m1-th occurrence of event X
6  n2 ← lower bound of the time distance from the
     m2-th occurrence of event Y
7  result ← []
8  if m1 = null && m2 = null then
9      result ← applyOriginalBetweenAnd(X, n1, Y, n2)
10 end
11 else
12     if m1 = null then m1 ← 1
13     if m2 = null then m2 ← 1
14     result.append(applySpecialBetweenAnd(m1, X, n1,
         m2, Y, n2))
15 end
```

returns a lists of trace segments. We assume the parameter *scope* to have the form "`between [m1] X [at least n1 tu] and [m2] Y [at least n2 tu]`".

The function `applyBetweenAnd` starts by reading the parameters from the instance of the *between-and* scope (lines 1–6): variables X and Y correspond to the event names of the left and right scope boundaries; *m1* and *m2* represent the (optional) index of the specific occurrence of event X and event Y referred to in the scope definition; *n1* and *n2* are the (optional) lower bounds on the time distances from the two scope boundaries. Optional parameters are initialized to null if they are not defined. The output variable *result* is initialized to an empty list.

If both *m1* and *m2* are not defined, we compute the return value by calling the auxiliary function `applyOriginalBetweenAnd` (line 9), which retrieves all the trace segments delimited by the two boundary events (taking into account the distances from the boundaries, if defined). Otherwise, if either *m1* or *m2* is undefined, we compute the return value by calling the auxiliary function `applySpecialBetweenAnd` (line 14), which retrieves only one trace segment, as determined by the specific occurrences of the boundary events and by the time distance from the scope boundaries (if defined). Notice that in the latter case we consider as boundary the first occurrence of event X or Y (see assignments at lines 12–13).

Function `applyOriginalBetweenAnd` is shown in Algorithm 3. It takes in input the parameters X, Y, *n1*, *n2* of a *between-end* scope of the form "`between` X `[at least` $n1$ `tu] and` Y `[at least` $n2$ `tu]`" and returns a list of the trace segments determined by the scope semantics. The function goes through all the elements of the list and identifies all the segments delimited by the events X and Y, taking into account the parameters for the time distance from the scope boundaries.

Besides the output variable *result*, we define an integer tuple (i_1, t_1) to keep track of the starting point of a trace segment. More precisely, element i_1 refers to the index of the trace element that comes after the left bound of the segment (characterized by an occurrence of event X), while element t_1 points to the instant that is *n1* time units after the occurrence of the left bound of the segment. The tuple (i_2, t_2) is defined in a similar way, to keep track of the end point of a trace segment (characterized by an occurrence of event Y).

At each iteration of the loop (lines 5–24), for each element of the trace, the function first increments the variable *index* and assigns the event of the trace element to variable e as well as its timestamp to variable t (lines 6–8). Within the loop, a value of i_1 equal to 0 means that the left bound of the segment has not been found yet. When the current event matches X (line 10), i_1 is assigned the next index of the current event; t_1 is assigned the value of the timestamp of the current event incremented by *n1* time units (line 11). When variable i_1 is different than 0, it means that the left boundary has been found while the right boundary has not been found yet. In this case, the function keeps scanning the remaining trace elements until it finds an occurrence of event Y. If the current event matches Y and if the current index is more than i_1 (line 14), i_2 is assigned the previous index of the current event; t_2 is assigned the value of the timestamp of the current event decremented by *n2* time

Algorithm 3: applyOriginalBetweenAnd

Input: strings X, Y and integers $n1, n2$ ($n1 = 0$, $n2 = 0$ by default), i.e., the parameters of a *between-and* scope structured as "`between` X `[at least` $n1$ `tu] and` Y `[at least` $n2$ `tu]`"

Output: *result* : a list of trace segments, as determined by the parameters of the scope

1 $result \leftarrow []$
2 $index \leftarrow 0$
3 $(i_1, t_1) \leftarrow (0, 0)$
4 $(i_2, t_2) \leftarrow (0, 0)$
5 **for** $elem \in self.traceElements$ **do**
6     $index \leftarrow index + 1$
7     $e \leftarrow elem.event$
8     $t \leftarrow elem.timestamp$
9   **if** $i_1 = 0$ **then**
10   **if** $e = X$ **then**
11             $(i_1, t_1) \leftarrow (index + 1, t + n1)$
12   **end**
13   **end**
14   **else if** $e = Y$ && $index > i_1$ **then**
15         $(i_2, t_2) \leftarrow (index - 1, t - n2)$
16         $segment \leftarrow self.traceElements[i_1 .. i_2]$
17   **if** $n1 \neq 0 \parallel n2 \neq 0$ **then**
18             $segment \leftarrow$ trace elements in *segment* with timestamps t' satisfying $t_1 \leq t' \leq t_2$
19   **end**
20   *result*.`append(`*segment*`)`
21         $(i_1, t_1) \leftarrow (0, 0)$
22         $(i_2, t_2) \leftarrow (0, 0)$
23   **end**
24 **end**
25 **return** *result*

units (line 15). At this point, the function extracts a trace segment comprised between indexes i_1 and i_2 (line 16), whose trace elements have a timestamp comprised between t_1 and t_2 (line 18). This segment is added to the output variable *result* and then the tuples (i_1, t_1) and (i_2, t_2) are reset (for the next loop iteration).

Function `applySpecialBetweenAnd` (not shown here) is defined similarly to function `applyOriginalBetweenAnd`, but is extended with two additional parameters *m1* and *m2*, referring to the specific index of the occurrence of each of the two boundary events. This function identifies a single segment of the trace between the *m1*-th occurrence of event X and the *m2*-th occurrence of event Y, taking into account the constraints on the time distances from the two scope boundaries. The function body is similar to that in Algorithm 3 and is extended with a counter that keeps track of the number of occurrences of a boundary event found while traversing the trace elements. Since only one segment has to be identified with this function, the main loop is interrupted as soon as such a segment is found.

6.4 OCL functions for patterns

In this section we present two examples of OCL functions that are used to check if a pattern holds on a sub-trace. We show the pseudocode of functions `checkPatternExistence` and `checkPatternPrecedence`. These functions take as input a sub-trace and an object representing a pattern

Algorithm 4: checkPatternExistence

Input: a trace segment *subtrace* and an instance of the *existence* pattern *pattern*, in the form "`eventually [`op n`]` E"

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

1 $E \leftarrow$ event name in *pattern*
2 $op \leftarrow$ comparison operator of the bound on the number of occurrences of event E
3 $n \leftarrow$ threshold of the occurrence number of event E
4 $count \leftarrow$ the number of occurrences of event E in *subtrace*
5 **return** `compare(`*count*, *op*, *n*`)`

in *TemPsy*, and return whether the pattern holds on the input sub-trace.

6.4.1 Existence

Function `checkPatternExistence` (see Algorithm 4) takes in input a trace segment (denoted by the variable *subtrace*) and an instance of the *existence* pattern (denoted by the variable *pattern*). First, the function retrieves some parameters from variable *pattern*: the event name *E*, the comparison operator *op*, and the threshold on the number of event occurrences n (lines 1–3). Then, variable *count* is set to the number of occurrences of event E in the input *subtrace* (line 4). The function returns the result of the invocation of the auxiliary function `compare`, which compares the value of *count* against the value of parameter n using the comparison operation defined by *op* (which can be "`at least`", "`at most`", or "`exactly`"). The auxiliary function `compare`, not shown here for space reasons, takes into account also the case in which *op* is null, meaning that the function returns true if the value of *count* is greater than 0.

6.4.2 Precedence

The definition of function `checkPatternPrecedence` comes in four variants, to consider the case whether no time distance is specified between the two blocks of the patterns, and the three cases with the different comparison operators (i.e., "`at least`", "`at most`", and "`exactly`"). In the rest of this section we describe the function `checkPatternPrecedenceAtLeast`, shown in Algorithm 5; the functions for the other cases are similar and omitted for space reasons.

The function `checkPatternPrecedenceAtLeast` takes in input a trace segment and the parameters of an instance of a *precedence* pattern: $block_1$, $block_2$, and the optional time distance n between them. Notice that $block_1$ and $block_2$ can be either an atomic event or a chain of events with optional constraints on the time distances in between.

The semantics of the pattern prescribes that each occurrence of $block_2$ is preceded, possibly with a certain time distance, by an occurrence of $block_1$. In practice, we need to check whether there is an occurrence of $block_1$ before the *first* occurrence of $block_2$ (and at a certain time distance, if required), since this implies that any other occurrence of $block_2$ occurring after the first one is preceded by an occurrence of $block_1$. We report a violation if we cannot find an occurrence of $block_1$ before the first occurrence of $block_2$ or if the distance between the two blocks is less than n.

Algorithm 5: checkPatternPrecedenceAtLeast

Input: a trace segment *subtrace* and the parameters of an instance of *precedence* pattern of the form "$block_1$ `preceding [at least n tu]` $block_2$": two events (chains) $block_1$ and $block_2$, and a threshold n (n=1 by default) of the time distance between $block_1$ and $block_2$

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

1 $size_1, size_2 \leftarrow$ the sizes of $block_1$ and $block_2$
2 $firstOfBlock1 \leftarrow block_1$.`first()`.$event$
3 $firstOfBlock2 \leftarrow block_2$.`first()`.$event$
4 $flag_1 \leftarrow$ `true`
5 $(i_1, pt_1) \leftarrow (1, 0)$
6 $(i_2, pt_2) \leftarrow (1, 0)$
7 **for** $elem \in subtrace$ **do**
8   $e \leftarrow elem.event$
9   $t \leftarrow elem.timestamp$
10   **if** $flag_1$ **then**
11     $op \leftarrow block_1[i_1].timeDistance.op$
12     $t' \leftarrow pt_1 + block_1[i_1].timeDistance.value$
13   **if** $e = firstOfBlock1$ **then** $(i_1, pt_1) \leftarrow (2, t)$
14   **else if** $e = block_1[i_1].event$ && `compare(`t, op, t'`)` **then**
15       $(i_1, pt_1) \leftarrow (i_1 + 1, t)$
16   **if** $i_1 = size_1 + 1$ **then** $flag_1 \leftarrow$ `false`
17   **end**
18   **else** $(i_1, pt_1) \leftarrow (1, 0)$
19   **end**
20   $op \leftarrow block_2[i_2].timeDistance.op$
21   $t' \leftarrow pt_2 + block_2[i_2].timeDistance.value$
22   **if** $e = firstOfBlock2$ **then**
23   **if** $flag_1 \parallel t < pt_1 + n$ **then**
24   **if** $size_2 = 1$ **then return** `false`
25   **else** $(i_2, pt_2) \leftarrow (2, t)$
26   **end**
27   **else return** `true`
28   **end**
29   **else if** $e = block_2[i_2].event$ && `compare(`t, op, t'`)` **then**
30   **if** $i_2 = size_2$ **then return** `false`
31   **else** $(i_2, pt_2) \leftarrow (i_2 + 1, t)$
32   **end**
33   **else** $(i_2, pt_2) \leftarrow (1, 0)$
34 **end**
35 **return** `true`

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

The function contains a loop that iterates through all the elements of the input subtrace, trying to match each element with $block_1[i_1]$ (lines 10–19) and with $block_2[i_2]$ (lines 20–33). As for matching $block_1$, until $flag_1$ is `true`, the algorithm checks whether the current element is part of an occurrence of $block_1$. If it matches the first event of $block_1$ (line 13), the variable i_1 is set to 2 and pt_1 is updated with the current timestamp. Otherwise, if the current trace element is an occurrence of the event defined at $block_1[i_1]$ (with i_1 being greater than 1)) and the constraint on the distance (if defined[9]) from the previous event at $block_1[i_1 - 1]$ holds (line 14), index i_1 is incremented and variable pt_1 is updated with the timestamp of the current trace element (line 15). Moreover, if the matched event is the last event of $block_1$, variable $flag_1$ is set to `false` (line 16). Otherwise, the tuple (i_1, pt_1) is reset on line 18.

Within each single iteration of the loop, the algorithm also checks whether the current trace element is part of an occurrence of $block_2$. If the occurrence of the *first* event of $block_2$ is detected (line 22), there are two cases that may lead to a violation. Either $block_1$ has not been fully matched yet (i.e., variable $flag_1$ is `true`) or it has been fully matched but the timestamp of the current trace element (that matches the first element of $block_2$) violates the constraint on the distance between $block_1$ and $block_2$. If one of these two conditions holds (line 23), if $block_2$ is composed of only one event, a violation is reported (line 24), otherwise (line 25) the algorithm goes on to match[10] the rest of $block_2$ (lines 29–32), since the current element might actually not be part of an instance of $block_2$. If both of these conditions are not satisfied (line 27), it means that there is no violation, i.e., the first block has been fully matched and the distance constraint between the two blocks is satisfied; hence, there is no need to match[11] the remainder of $block_2$ and the algorithm returns `true`. If the occurrence of the *first* event of $block_2$ is not detected (line 29), if the current trace element is a match for the event at $block_2[i_2]$ (with i_2 being greater than 1) and the constraint on the distance (if defined) from the previous event at $block_2[i_2 - 1]$ holds, the algorithm either reports a violation when $block_2$ is fully matched (line 30) or moves the match one step further: the index i_2 is incremented by 1 and pt_2 is updated with the timestamp of the current trace element (line 31). If the current element is not part of an occurrence of $block_2$, the tuple (i_2, pt_2) is reset (line 33).

The algorithm returns `true` (line 35) when there is no violation reported in the loop.

6.5 The approach at work: an example

We now show how the approach works on a simple example. Consider the trace shown in Fig. 3 and the property "Event X shall happen at least twice before the third occurrence of event Y", which can be expressed in *TemPsy* as "`before 3` Y `eventually at least 2` X", using a *before* scope combined with an *existence* pattern.

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

[10] Notice that in this case a violation is reported only if *block*2 is fully matched (line 30).

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

Checking this property on the trace using our model-driven approach is reduced to the evaluation of the OCL invariant shown in Fig. 12; this evaluation goes as follows.

After extracting the scope and pattern from the property and assigning them to variables *scope* and *pattern* (line 3 in Fig. 12), function `applyScopeBefore` (detailed in Algorithm 1) is invoked to select the sub-traces determined by the parameters of *scope*. In this example, parameter m is 3, the event name X is "`Y`", and parameters *op* and n are undefined because the scope has no constraint on the time distance from the scope boundary.

The statement at line 7 of Algorithm 1 will determine the timestamp of the third occurrence of event Y (38 in this case) and assign it to variable t. Since the parameter *op* is undefined, the case statement at line 20 of the algorithm will be executed, selecting the sub-trace containing events with a timestamp less than or equal to 38, i.e., the sub-trace having the event X at timestamp 2 as first event and the event Y at timestamp 38 as last event. This sub-trace is the only element contained in the list returned by Algorithm 1.

The evaluation of the OCL invariant shown in Fig. 12 continues with the evaluation of the expression `subtraces->forAll(subtrace | checkPatternPrecedence(subtrace, pattern))`; in this case, variable *subtraces* contains the list returned by function `applyScopeBefore`, as discussed above. Function `checkPatternExistence` will be invoked once (because list *subtraces* contains only one element), taking in input the sub-trace and variable *pattern*, to check the pattern over the sub-trace. In this example, for Algorithm 4, the parameter corresponding to the event name E is "`X`", the comparison operator *op* is "at least", and the parameter n is 2. The execution of the statement at line 4 in Algorithm 4 will yield 3 in the variable *count*, since there are three occurrences of event X in the input sub-trace. Afterwards, the value of *count* is compared to the parameter n using the comparison operator *op*; in this case, the algorithm will return true (since $3 > 2$), indicating that the property is satisfied on this sub-trace.

Since there are no more sub-traces on which to apply function `checkPatternExistence`, the evaluation of the invariant will return *true*, indicating that the input property is not violated by the trace.

6.6 Tool Implementation

We have implemented our model-driven approach for trace checking of *TemPsy* properties in a tool named TemPsy-Check. The tool is based on Xtext [29] and Eclipse OCL; it is publicly available at `http://weidou.github.io/TemPsy-Check`.

TemPsy-Check takes in input a log file in CSV format and converts it to an intermediary representation (called "trace description"), defined as a domain-specific language using the Xtext framework. We have introduced this intermediate representations for traces to support, in the future, multiple input raw formats for trace logs. The trace description is then used to generate an XMI file corresponding to an instance of the trace model. The tool also takes in input a list of *TemPsy* properties (defined using the textual notation shown in Fig. 2) and converts them into an XMI-based format. The evaluation of the OCL constraints corresponding (as described in the previous subsections) to the properties to check on the trace is done using the OCL checker included in Eclipse OCL [28].

7 Evaluation

In this section we report on the evaluation of TemPsy-Check. The evaluation focuses on the scalability of the tool, to assess the relationship between the time taken to check a property on a trace and the structural properties of the trace (e.g., length, distribution of events) and the type of property to check. We also compare the performance of TemPsy-Check with a state-of-the-art alternative technology.

We have conducted our evaluation using a benchmark consisting of a subset of the properties extracted from the requirements specification documents of the eGovernment application developed by our public service partner, described in section 4. Out of the 47 properties documented in the case study, we left out of the benchmark the nine properties using the *after-until* pattern. Properties of this type can be rewritten using the *between-and* scope, possibly in conjunction with an *after* scope: for this reason, they would not have provided additional insights to our scalability analysis. The 38 properties used for the evaluation are listed in a sanitized form in Table 2. The actual textual description of each property has been omitted for confidentiality reasons; for each property we only detail its structure in terms of *scope* + *pattern*. The events involved in the property (e.g., "a citizen requests a certificate") are denoted using uppercase letters.

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

To synthesize these traces we implemented a trace generator program. This program allows for generating diverse (in terms of size, patterns, scopes, event positions and frequency) and realistic traces, without introducing bias. The generator takes in input a property, the desired length of the trace to generate and additional parameters depending on the type of property given in input and the factors one wants to control. To determine the position in the trace of the events occurring in the input property, the generator takes into account the temporal and timing constraints prescribed by the semantics of the scope and the pattern used in the property. Positions in the trace that are deemed not relevant for the evaluation of the property are filled with a dummy event. The details of the trace generation strategy depend on the scope and pattern used in the properties and are discussed in the next subsections.

As an additional contribution of the paper, we also make available in the TemPsy-Check GitHub repository the artifacts used in the evaluation, to contribute to the building of a public repository of case studies for evaluating trace checking/run-time verification procedures.

Table 2: *TemPsy* properties used for the evaluation

P1: globally always A
P2: globally never B
P3: globally eventually at least 2 A
P4: globally eventually at most 3 A
P5: globally A responding at most 1000 tu B
P6: globally A responding exactly 1000 tu B
P7: globally A preceding at most 6000 tu B
P8: globally A preceding at least 100 tu B
P9: globally A preceding exactly 100 tu B
P10: globally A, B preceding at least 1000 tu C, D
P11: globally A responding at least 1000 tu B, C
P12: globally A responding B
P13: before A eventually B
P14: before 3 A eventually at least 2 B
P15: before 2 A never B
P16: before A B responding at most 3000 tu C
P17: before A at least 1000 tu B responding at least 1000 tu C
P18: before A B, # at most 6000 tu C preceding D
P19: before 3 A B, # at least 1000 tu C preceding D
P20: before A B preceding C
P21: after A at most 5000 tu eventually B
P22: after A always B
P23: after 2 A exactly 5000 tu eventually B
P24: after A B responding at least 1000 tu C
P25: after A B preceding at most 3000 tu C, D
P26: after 2 A at most 3000 tu B preceding C, D
P27: after 2 A never B
P28: after A at most 1000 tu B responding at most 10 tu C
P29: after A B preceding at least 2000 tu C
P30: after A eventually at most 6 B
P31: after 2 A at least 5000 tu eventually B
P32: between A and B always C
P33: between A at least 1000 tu and B at least 500 tu never C
P34: between A and B C responding at least 1000 tu D
P35: between A and B never exactly 2 C
P36: between 3 A and B always C
P37: between A at least 1000 tu and 2 B C preceding at least 1000 tu D
P38: between 2 A and 2 B eventually at most 10 C

The next three subsections report on the checking of properties using, respectively, the *globally*, *before/after*, and *between-and* scopes. For each group of properties we first describe the trace generation strategy and then present and discuss the results. The section ends with a discussion of the results and of the threats to validity. Notice that out of the three types of scope considered for the evaluation, the properties using a *globally* scope represent 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 through the entire length of the trace.

Moreover, to assess scalability, we also need a baseline of comparison. Such baseline should be the best available tool that can be considered an alternative to TemPsy-Check. We identified such a tool among the participants to the "offline monitoring" track of the first international Competition on Software for Runtime Verification [8] (CSRV 2014), held in September 2014 as a satellite event of the 14th International conference on Runtime Verification (RV'14). Out of the four tools (RiTHM2 [52], MonPoly [9], StePr, QEA [7]) qualified for the final round of the competition, RiTHM2 and StePr were not publicly available[12] at the time of writing. Between the remaining two, we chose MonPoly over QEA because only the former supports a real specification language (MFOTL, a metric first-order temporal logic) that is conceptually close to *TemPsy*. On the other hand, QEA does not support any input language and uses an automata-based formalism: the user has to write a Java program that builds the automaton corresponding to the property to check. To perform the comparison with MonPoly, we manually translated the properties into MFOTL formulae; these formulae are also available in the TemPsy-Check GitHub repository. We remark that our goal, in this comparison, is not to fare better than existing technology, but to verify that an MDE approach to offline trace checking is viable from a scalability standpoint.

[12] The first version of RiTHM is available but it only supports run-time verification of C programs. As for StePr, no reference is available in the competition report [8] or online.

The results reported in this section have been measured on a desktop computer with a 3 GHz Intel Dual-Core i7 CPU and 16GB of memory, running Eclipse DSL Tools v. 4.6.0M3 (Neon Milestone 3), JavaSE-1.7 (Java SE v. 1.8.0_25-b17, Java HotSpot (TM) 64-Bit Server VM v. 25.25-b02, mixed mode), Eclipse OCL v. 6.0.1, and MonPoly v. 1.1.6. All measurements reported correspond to the average value over 100 runs of the check procedure (on the same trace, for the same property).

7.1 Properties using the *Globally* scope

Properties defined with the *globally* scope are the most important for assessing the scalability of our approach with respect to the trace length. Indeed, the semantics of this scope requires the tool to check the property pattern through the *entire* trace, while in the case of the other scopes, property patterns are checked only on some segments of the input trace (i.e., on sub-traces). In our collection of properties there are 12 using the scope *globally*, in combination with various patterns; they correspond to properties P1–P12 listed in Table 2.

For this type of properties, given that they are the most challenging in terms of scalability, we address the following research questions:

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

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

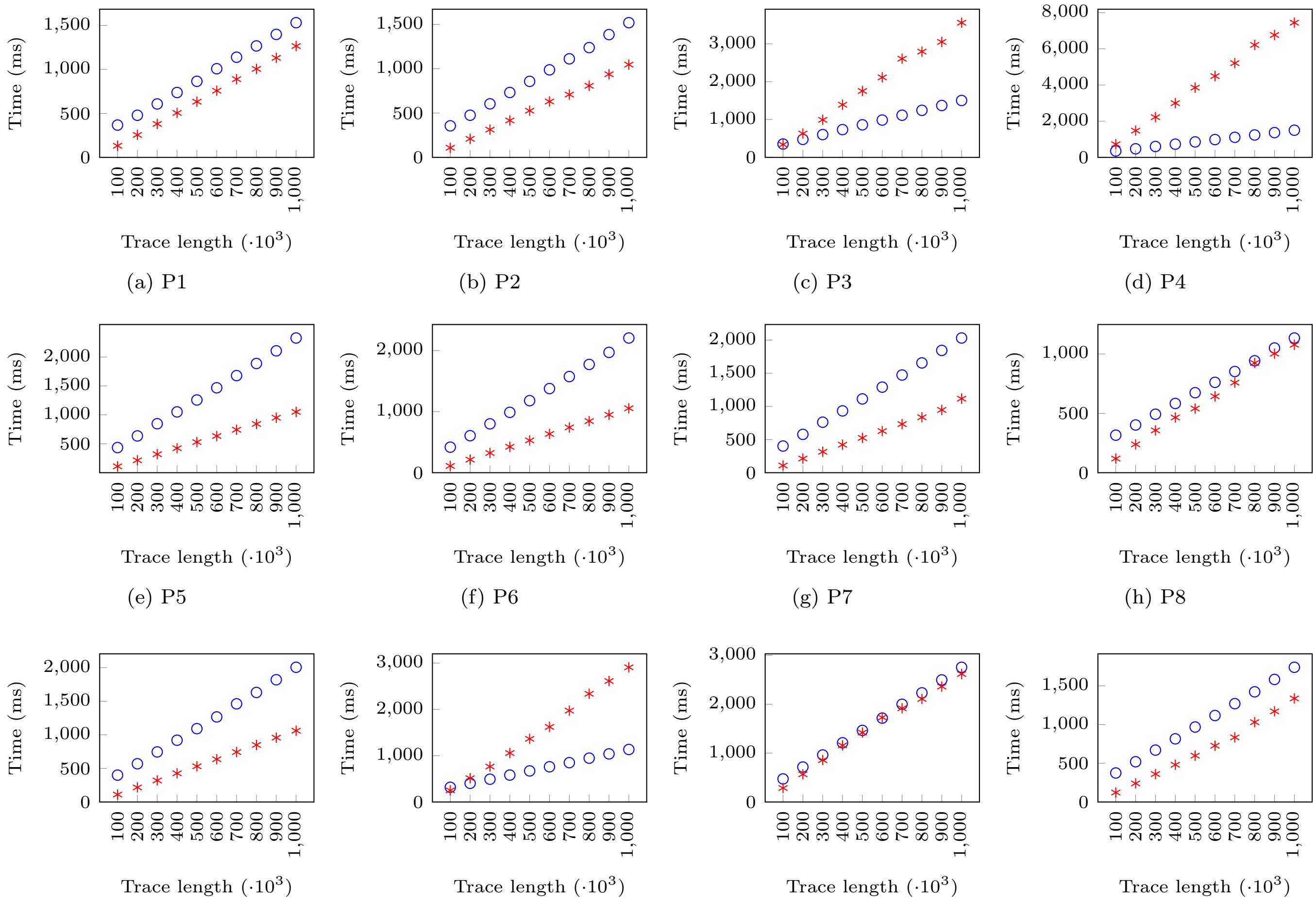
RQ-G3) How does TemPsy-Check compare with MonPoly in terms of execution time?

7.1.1 Trace Generation Strategy

In the case of the *globally* scope the generation of the trace is determined only by the semantics of the pattern used in the property.

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

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



(a) P1 (b) P2 (c) P3 (d) P4

(e) P5 (f) P6 (g) P7 (h) P8

(i) P9 (j) P10 (k) P11 (l) P12

Fig. 13: Comparison between the execution time of TemPsy-Check (○) and of MonPoly (∗) for properties with the *globally* scope

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

In the case of a property containing a *precedence* or *response* pattern, we generate a number of occurrences of events (involved in the property) equal to 10% of the length of the trace. This value has been selected based on the frequency of events observed in the application whose requirements are expressed through the properties shown in Table 2. The simplest case is for a property like `globally B responding at most 10 tu A`: assuming a trace length of 1M, we would generate 50K occurrences of the pattern (i.e., pairs of `A` and `B`), for a total of 100K occurrences of `A` and `B`. More complex cases have to take into account the event chains used in the property. For the distribution of the occurrences of the pattern across the trace we have to consider the distance between events. For example, for the property aforementioned, each occurrence of the response pattern would span over at most 10 time units; this is the maximum distance between an occurrence of `A` and the corresponding occurrence of `B`. The number of pattern occurrences to generate and the maximum time span of each pattern occurrence are the parameters used to randomly allot the pattern occurrences over the trace, according to a uniform distribution.

7.1.2 Evaluation

We run the trace checking procedure for properties P1–P12; each property was checked on ten different traces, with length (i.e., number of events) varying from 100K to 1M. The twelve plots in Fig. 13 depict the execution time of TemPsy-Check (denoted by ○) and of MonPoly (denoted by ∗) for each of the properties P1–P12, for different trace lengths. The execution time for both tools has been measured using the `time` Unix command.

We answer RQ-G1 by observing that the time taken by TemPsy-Check ranges from about one hundred milliseconds to a bit more than two seconds, and increases linearly with the length of the trace, depending on the type of property. To answer RQ-G2, the results show that the properties more taxing on the execution time are those with a *response* or *precedence* pattern (e.g., P5, P6, P7, P9, P11). Regarding RQ-G3, we observe that the time taken by MonPoly ranges from about one hundred milliseconds to a bit less than eight seconds, and is also linear with respect to the length of the trace. MonPoly takes longer for checking properties with a *(bounded) existence* pattern (e.g., P3, P4) and with a *precedence* pattern that contains a distance constraint of type "`at least`" (e.g., P10). We can answer RQ-G3 stating that, except for the case of properties P3, P4, and P10, the two tools perform almost similarly, with absolute differences between execution times that are quite small (less than one second). In the case

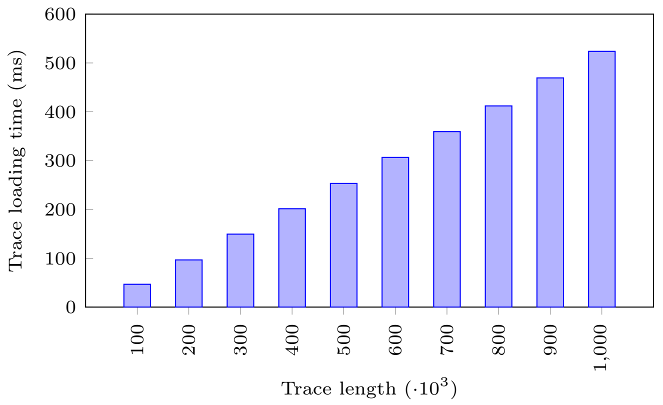


Fig. 14: Trace loading time of TEMPSY-CHECK for traces with various lengths

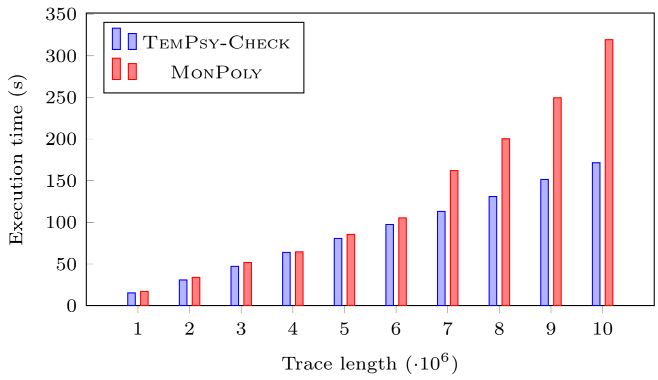


Fig. 15: Comparison of the execution time for the batch checking of ten properties with the *globally* scope

of properties P3, P4, and P10, TEMPSY-CHECK performs much better than MONPOLY. A possible explanation for the slower time of MONPOLY for these properties could be the structure of the corresponding MFOTL formulae, which contain several nested temporal operators to express the "eventually at least/at most" pattern (P3, P4) and an event chain (P10).

The execution times discussed above include not only the time to perform the actual check, but also the time to parse/load the trace to check[13]. As shown in Fig. 14, the average trace loading time for TEMPSY-CHECK, measured through instrumentation, ranges from 55 ms to 550 ms, growing linearly for various trace lengths. Notice that for checking a single property on a trace with TEMPSY-CHECK, the trace loading time can take, for larger traces, from one-fourth to one-third of the total execution time. Although these values for the trace loading time can seem high, we expect the loading time not to impact on the total execution time in the case of *batch property checking*, i.e., checking multiple properties at the same time on a trace. Checking in batch mode a set of properties, rather than individual ones, is common in enterprise scenarios in which, for example, the set of properties to check is decided by the entity that has invoked a business process [5].

To further investigate this aspect, we compared the execution time of TEMPSY-CHECK and MONPOLY for batch checking ten properties (P3–P12), over ten traces, with length ranging from 1M to 10M. These traces have been obtained by concatenating the traces used for the experiment described above, and by renaming the events within each trace being concatenated, to avoid name clashes. We executed TEMPSY-CHECK by providing in input the list of the ten properties to check. We executed MONPOLY by providing in input one formula corresponding to the conjunction of the ten formulae equivalent to properties P3–P12. Figure 15 shows the result of the comparison: the performance of the two tools are similar for traces of length up to six millions; over this threshold, MONPOLY gets slower.

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

7.2 Properties using the *Before/After* scopes

Properties defined using the *before/after* scopes, differently from the ones using a *globally* scope, have to be checked only on the portion of the trace delimited by the scope boundary. Hence, their scalability does not relate in a direct way with the length of the trace. Nevertheless, they can help us assess whether and how the type of property (e.g., the scope used within the property) impacts on the total execution time. We have checked eight properties with the *before* scope (properties P13–P20 in Table 2) and eleven properties with the *after* scope (properties P21–P31 in Table 2).

For this type of properties, to assess how the type of scope used in them impacts on the total execution time, we address the following research questions:

RQ-BEAF1) What is the relation between the time to compute the boundary of the scope and the position of the boundary?

RQ-BEAF2) What are the types of scope most expensive to compute?

Notice that we do not compare with MONPOLY since the concept of "scope" is not a first-class object in MFOTL formulae.

7.2.1 Trace Generation Strategy

As remarked above, for this type of properties the scalability of the checking procedure does not relate in a direct way with the length of the trace. Hence, for both types of scopes, we fix the length of the generated trace to 100K. To answer the research questions above, we vary the length of the sub-trace as determined by the scope boundary, i.e., we vary the position of the boundary event in the trace. In the case of properties with a *before* scope, the boundary event is placed in positions from 10K to 100K, with a 10K step increment; similarly, for properties with an *after* scope, the position of the boundary event varies from 10K to 90K, with a 10K step increment.

For properties referring to a specific occurrence of an event in their scope part, such as `before 3 B`... or `after 4 A`..., we only control the position of the actual scope boundary (e.g., the third occurrence of `B` or the fourth occurrence of `A` in the examples above). The other previous occurrences of the boundary event

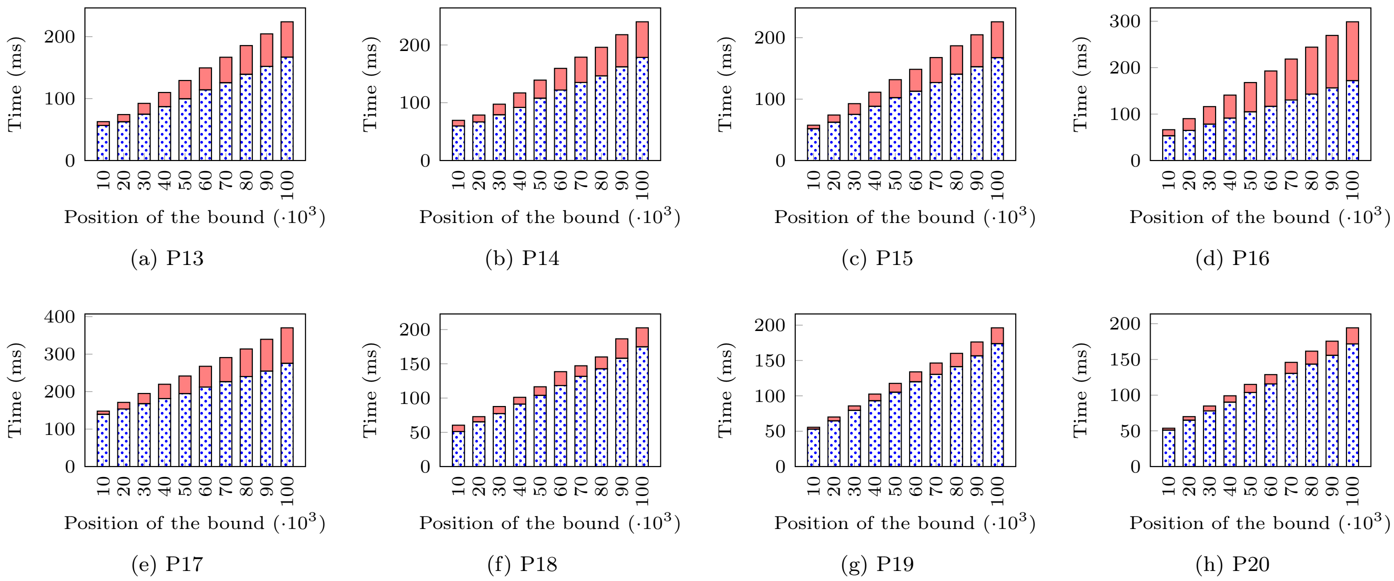


Fig. 16: Scope time and pattern time of TEMPSY-CHECK for checking properties with a *before* scope

are generated in random positions using a uniform distribution ove the range $[0, \text{position of the boundary}]$ (for properties with a *before* scope), and over the range $[\text{position of the boundary}, \text{trace length}]$ (for properties with an *after* scope).

The generation of the patterns corresponding to the actual properties follows the steps described in section 7.1.1.

7.2.2 Evaluation

We instrumented TEMPSY-CHECK to report the time taken to compute the boundary of a scope (i.e., to determine the sub-trace on which to check each property pattern), hereafter referred to as *scope time*, as well as the time to check the pattern on the sub-trace, hereafter referred to as *pattern time*. More specifically, scope time corresponds to the time taken to evaluate expressions of type `applyScope*S*` in Fig. 12, while pattern time corresponds to the time taken to evaluate expressions of type `checkPattern*P*` in Fig. 12.

Figures 16 and 17 show the scope time (denoted by) and the pattern time (denoted by) for checking, respectively, properties P13–P20 (with a *before* scope) and property P21–P31 (with an *after* scope), when varying the position of the scope boundary. Notice that while in the case of a *before* scope a higher position of the bound corresponds to a longer length of the sub-trace, in the case of an *after* scope a lower position of the bounds corresponds to a longer length.

To answer RQ-BEAF1, we observe from the plots that both in the case of the *before* scope and in the case of the *after* scope, the scope time grows linear with respect to the position of the scope boundary. This is due to the increase of the length of the sub-trace delimited by the scope boundary.

We answer RQ-BEAF2 by looking at the scope time for properties P17, P21, P23, P26, P28, P31. These properties are the most taxing in terms of scope time because the scope boundary is defined with a distance constraint. This is particularly true for the cases in which the boundary is defined using an "`at most`" constraint (see P21, P26, and P28).

7.3 Properties using the *Between-and* scope

Properties with a *between-and* scope, similarly to the ones with a *before*/*after* scope, are checked on a portion of trace provided in input. Depending on the variant of this scope, the portion of the trace on which properties are checked might include one or more segments. The scopes used in properties P32–P35 can potentially select multiple segments on a trace, while the scopes in properties P36–P38 select exactly one segment on a trace, as determined by the specific event occurrence used in the scope boundaries (e.g., as in the case of `between 3 A and 2 B`).

For this type of properties, given the two variants of the *between-and* scope, we address the following research questions:

RQ-BA1) For the scope variant that can select multiple segments on the trace, given a fixed length for the segments, what is the relation between the number of segments and the time to compute the scope?

RQ-BA2) For the scope variant that can select multiple segments on the trace, given a fixed number of segments, what is the relation between the length of the segment and the time to compute the scope?

RQ-BA3) For the scope variant that can select only a single segment, given a fixed length for this segment, what is the relation between the position of the segment and the time to compute the scope?

RQ-BA4) For the scope variant that can select only a single segment, given a fixed position of this segment, what is the relation between the length of the segment and the time to compute the scope?

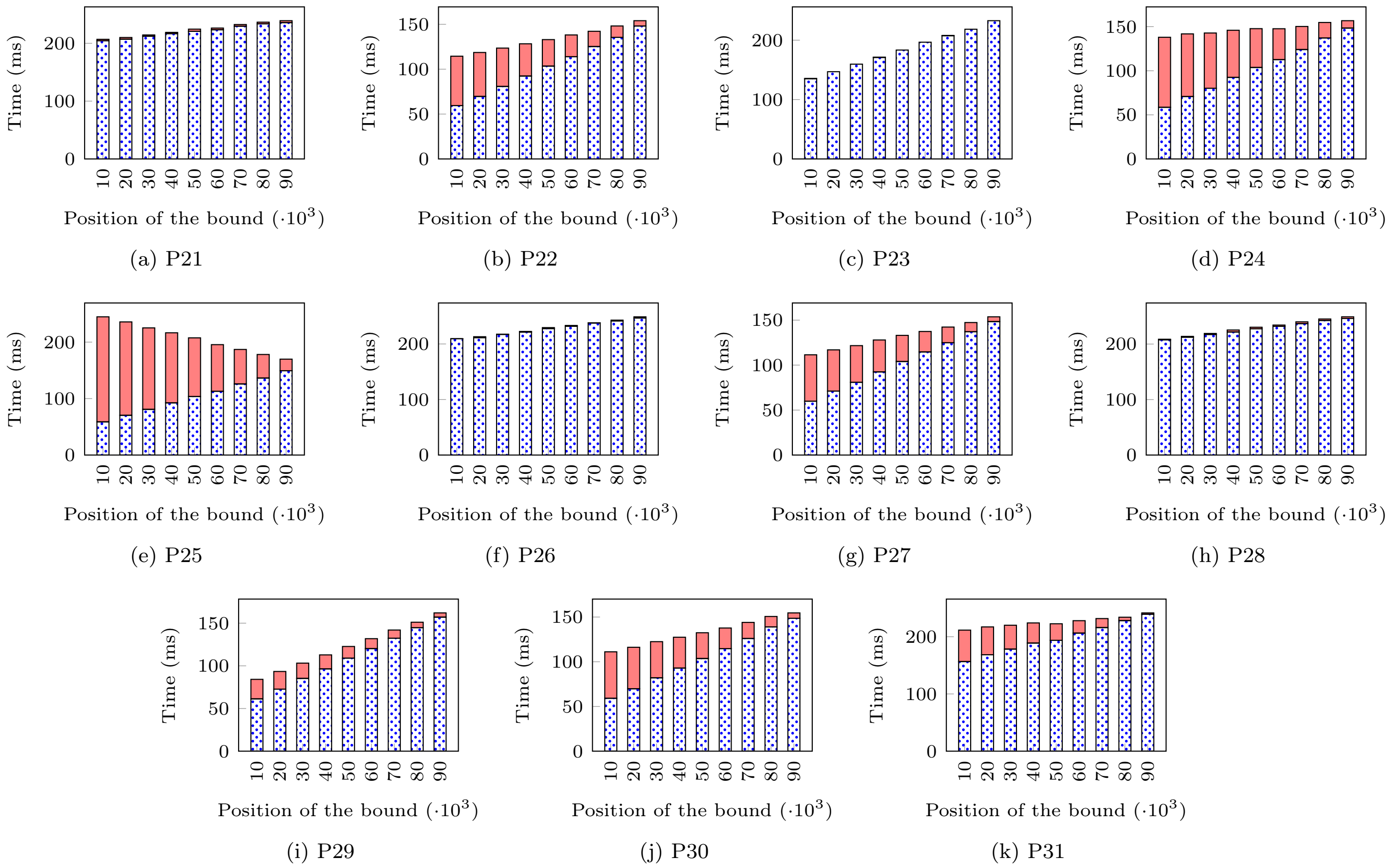


Fig. 17: Scope time and pattern time of TemPsy-Check for checking properties with an *after* scope

Notice that also in this case we do not compare with MonPoly because the concept of "scope" is not a first-class object in MFOTL formulae.

7.3.1 Trace Generation Strategy

For both types of *between-and* scope variants, we fix the length of the generated trace to 100K. To answer RQ-BA1 and RQ-BA2 we consider properties P32–P35. For these properties, we control two parameters for the trace generation: the length L of each segment selected by the scope and the number of segments N. By fixing L to 2000, we can split the 100K trace into 50 segments. The generator varies the number N of actual segments to generate from 5 to 50, with a 5-step increment. By fixing N to 20, and assuming a minimum length of 2000 for a segment (given the time constraints in P33), the generator produces traces with segments of length varying from 2000 to 5000, with 1K-step increment.

To answer RQ-BA3 and RQ-BA4 we consider properties P36–P38. For these properties we control two parameters: the length L' of the segment and the position P of one of its bounds. By fixing L' to 10K, we vary the position of the right bound from position 10K to position 100K with 10K-step increment, i.e., we vary the position of the segment in the trace. By fixing the position P to 10001, we can vary L' from 10000 to 90000, with 10K-step increments.

7.3.2 Evaluation

As done above for the case of properties with a *before/after* scope, we also distinguish between scope time and pattern time for checking properties with a *between-and* scope.

To answer RQ-BA1 we observe the plot in Fig. 18. The scope time for properties P32–P35 when varying the number of segments (as determined by the scope) on which to check the property pattern, slightly increases with the number of segments to consider; the higher scope time for property P33 is due to the presence of a time distance constraint for the (left) scope boundary.

We answer RQ-BA2 by looking at the plot in Fig. 19. In the case of checking properties P32–P35 when fixing the number of segments to 20 and varying the segment length from 2000 to 5000, the scope time is almost constant (about 200 ms) for all properties but P33, because of the time distance constraint for the (left) scope boundary.

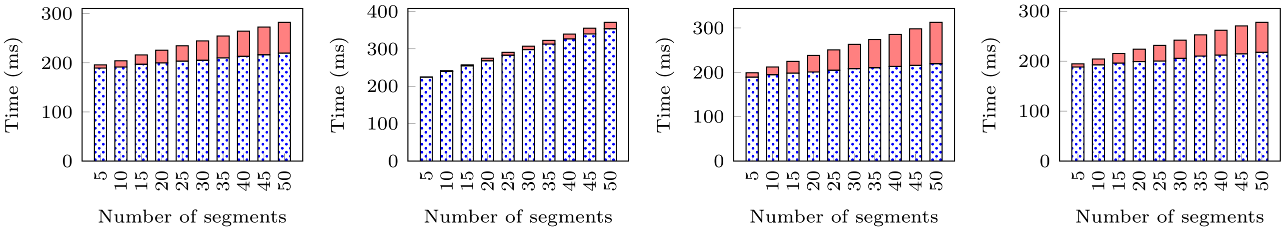
The answer to RQ-BA3 can be found by looking at the plot in Fig. 20. In the case of checking properties P36–P38 when varying the position of the segment on which the property pattern is checked and keeping the segment length constant, the scope time increases linearly with respect to the position of the segment.

We answer RQ-BA4 by observing the plot in Fig. 21. In the case of checking properties P36–P38 when varying the length of the segment, the scope time increases linearly with respect to the length of the segment.

7.4 Discussion

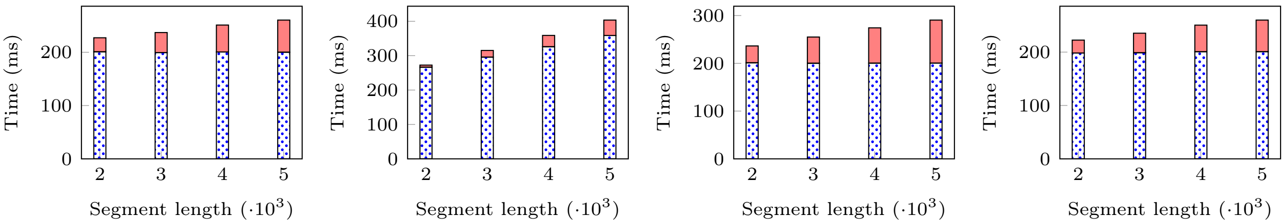
The results presented in the previous subsections have shown the feasibility of applying our model-driven approach for offline trace checking in realistic settings.

Our TemPsy-Check tool is a viable technology from a performance standpoint point as it can analyze very large traces (with one million events) in about two seconds. The



(a) P32 (b) P33 (c) P34 (d) P35

Fig. 18: Scope time and pattern time of TEMPSY-CHECK for checking properties with a *between-and* scope (multiple segments, fixed length)



(a) P32 (b) P33 (c) P34 (d) P35

Fig. 19: Scope time and pattern time of TEMPSY-CHECK for checking properties with a *between-and* scope (fixed number of segments, various lengths)

tool scales linearly with respect to the length of the input trace to check. Notice that "the input trace to check" can correspond also to a sub-trace of an actual, larger execution trace. This can be the case for properties referring to events occurring in time windows (see, for example, the service provisioning patterns presented in [15]). In these cases, one can first isolate from the original trace the window of interest and then feed the latter to our tool.

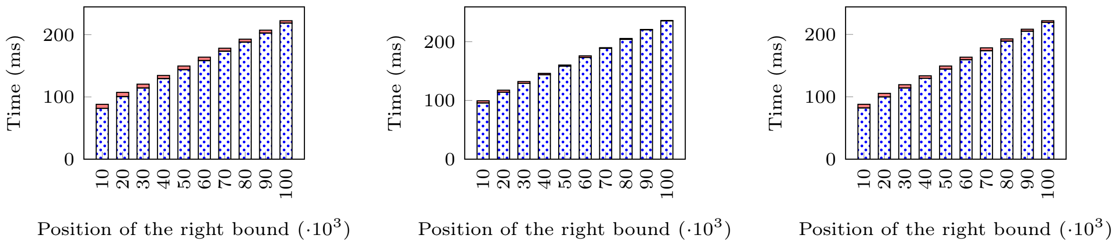
We have also compared the performance of our implementation to MONPOLY, a comparable, state-of-art tool. Despite the fact that MONPOLY is a tool that implements a dedicated algorithm [11] for trace checking of temporal logic properties, our TEMPSY-CHECK tool (which relies on a generalist OCL checker) not only achieves similar results, but in some cases it also performs better than MONPOLY.

We also remark that writing some of the properties in MFOTL was challenging (despite previous knowledge of MFOTL), much more than when using *TemPsy*. This challenge could be overcome by defining properties in *TemPsy* and then providing an automatic translation to MFOTL formulae or, dually, by building a system of property specification patterns on top of MFOTL. In both cases, one would have satisfied one of our requirements (R1, see section 1) and could have then relied on MONPOLY for trace checking. While this could be in principle a viable approach, it would not fulfill another requirement (R2, see section 1), which entails to rely on standard and stable MDE technology for checking temporal properties. We remark that these requirements are not specific to this project, but are more general because 1) analysts may not be able to handle the mathematical background required by temporal logic; and 2) there are many contexts where solutions have to be engineered by using standardized MDE technologies.

Overall, we can conclude that a model-driven approach to offline trace checking of realistic temporal properties is viable, even on very large traces, and compares favorably with the state-of-the-art.

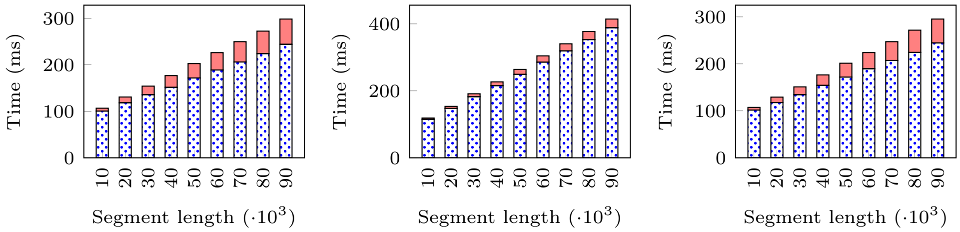
7.4.1 Threats to validity

The main threat to validity to the results presented above is the intrinsic presence of errors in the toolchain we developed. We tried to compensate for this by thoroughly testing the checker with traces and properties for which the oracle was previously known. Another potential threat is the fact that we have performed trace checking on *synthesized* traces. Real execution traces might be different, in terms of events occurrences and time distances. However, this threat does not affect our research question on scalability, as we want to analyze the execution time as a function of a number of parameters (e.g., trace length), while varying randomly other aspects (e.g., position of certain events). As explained at the beginning of this section, for that purpose, synthesized traces are better than real ones as they guarantee we have the data to perform our analysis by controlling certain factors and varying others randomly. Nevertheless, real traces (with faults in the system) could be helpful to assess the cost-benefit of the proposed trace checking procedure; this is out of the scope of this paper. Finally, as for the comparison with MONPOLY, we remark that its specification language (MFOTL) is more expressive than *TemPsy* (see also section 3.6), hence the performance of MONPOLY could have been negatively affected by the more complex implementation needed to support a richer specification language. Moreover, the MFOTL properties that we wrote to perform the comparison described in subsection 7.1 could be written in a different, but semantically-equivalent form that could lead to different results. We tried to mitigate this aspect by having the MFOTL formulae written by a person with ten years of experience in formal specification (and verification) with temporal logics. Furthermore, we believe that in practice, it might be hard



(a) P36 (b) P37 (c) P38

Fig. 20: Scope time and pattern time of TEMPSY-CHECK for checking properties with a *between-and* scope (single segment of fixed length, different positions)



(a) P36 (b) P37 (c) P38

Fig. 21: Scope time and pattern time of TEMPSY-CHECK for checking properties with a *between-and* scope (single segment, various lengths)

anyway for practitioners (with limited background in temporal logic) to find out what is the optimal way to express a property in MFOTL.

8 Related Work

The work presented in this paper is related to MDE approaches for specifying temporal properties and to approaches for trace checking/run-time verification. We review these areas in the next two subsections.

8.1 MDE approaches for specifying temporal properties

There have been several proposals in the MDE community to define high-level specification languages for expressing temporal properties; all these proposals are realized as temporal extensions of OCL. In the rest of this section we summarize them and discuss their differences and limitations with respect to *TemPsy*.

8.1.1 Pattern-based temporal extensions of OCL

The approaches that are most similar to *TemPsy* are those that extend OCL with support for Dwyer et al.'s property specification patterns.

Flake and Mueller [34] define a state-oriented OCL extension for expressing Dwyer et al.'s patterns over UML Statecharts configurations. The extension is based on the introduction of a special temporal operation, which can be applied to objects that have an associated Statechart. The evaluation of this operation at a certain time point yields the set of state configuration sequences in the time interval defined by the parameters of the operation. The extension, in addition to allowing for expressing the original definition of patterns in [27], adds also the support for specifying time distances in order patterns.

Table 3: Comparison between pattern-based temporal extensions of OCL and *TemPsy*

Language	Features				Tool
	NOOP	TDOP	SOS	TDS	support
[34]	-	+	-	-	-
[45]	-	-	-	*	-
[57]	+	*	-	*	n/a
[42]	+	*	-	*	+
TemPsy	+	+	+	+	+

Legend. NOOP: Number of Occurrences in *occurrence* Patterns; TDOP: Time Distance in *order* Patterns; SOS: Specific Occurrence in Scopes; TDS: Time Distance in Scopes; *: partial support; n/a: tool mentioned in the paper but not available.

Küster-Filipe and Anderson [45] propose a liveness template for OCL to define future-oriented time-bounded constraints that are expressed with a time-bounded *after* scope and an *existence* pattern. This template is defined in terms of the real-time temporal logic of knowledge, interpreted over timed automata, to allow for formal reasoning. The expressiveness of this extension is very limited since it supports only one scope/pattern combination.

Robinson [57] presents a temporal extension of OCL called $\mathrm{OCL_{TM}}$, developed in the context of a framework for monitoring of requirements expressed using a goal model. $\mathrm{OCL_{TM}}$ includes all the operators corresponding to standard LTL modalities, and supports Dwyer et al.'s patterns and time distances in patterns. In this regard, it is very close to the expressiveness of *TemPsy*, though it supports neither the reference to a specific occurrence of an event in scopes nor two types of constraints (as *TemPsy* does with the keywords '`at least`' and '`exactly`') on time distances in scopes and *order* patterns.

Kanso and Taha [42] introduce Temporal OCL, a pattern-based temporal extension of OCL. Although the support for temporal patterns is very similar between the two languages, Temporal OCL does not allow references to specific event occurrences in scope boundaries and does not fully support constraints on the time distance from a scope boundary (it only supports state-change events).

Table 3 provides a comparison of these four approaches with *TemPsy*, in terms of the following language features, derived from the analysis of the requirements specifications of our case study (see section 3.1): 1) the possibility of referring to the number of occurrences of an event in *occurrence* patterns (NOOP); 2) the possibility of defining a time distance between events in *order* patterns (TDOP); 3) the possibility of referring to a specific occurrence of an event in scopes (SOS); 4) the possibility of defining a constraint on the time distance from scope boundaries (TDS). The table also indicates whether the proposed language extension includes tool support.

As you can see, *TemPsy* is the only pattern-based language that provides support for all the specific features needed for the specification of requirements in the context of our case study.

8.1.2 Other temporal extensions of OCL

Temporal extensions of OCL that are not pattern-based are mainly realized by extending the language with temporal operators borrowed from standard temporal logic, such as "always", "until", "eventually", "next". A preliminary work in this direction appeared in [23]. OCL/RT [22] extends OCL with the notion of timestamped events (based on the original UML abstract meta-class `Event`) and two temporal operators, "always" and "sometimes". Events are associated with instances of classifiers and, by means of a special satisfaction operator, it is possible to evaluate an expression at the time instant when a certain event occurred. The OCL/RT extension allows for expressing real-time deadline and timeout constraints but requires to reason explicitly at the lowest-level of abstraction, in terms of time instants. Lavazza et al. [46] define the Object Temporal Logic (OTL), which allows users to write temporal constraints on Real-time UML (UML-RT) models. In particular, it supports the concepts of *Time*, *Duration*, and *Interval* to specify the time distance between events. Nevertheless, the language is modeled after the TRIO temporal logic [50], and the properties are written using a low level of abstraction. Ziemann and Gogolla [62] propose TOCL, an extension of OCL with LTL operators, to specify constraints on the temporal evolution of the system states. Being based on LTL, TOCL does not support real-time constraints. Bill et al. [18] define cOCL, an extension of OCL with CTL temporal operators to express properties over the lifetime of an instance model. These properties are then verified with an explicit state space model checking framework. Being based on CTL, cOCL does not support real-time constraints. The work on Flake and Mueller [33] goes in a similar direction, proposing an extension of OCL that allows for the specification of past- and future-oriented time-bounded constraints. They do not support event-based specifications; moreover, the proposed mapping into Clocked LTL does not allow to rely on standard OCL tools. Soden and Eichler [59] propose Linear Temporal OCL (LT-OCL) for languages defined over MOF meta-models in conjunction with operational semantics. LT-OCL contains the standard LTL operators. The interpretation of LT-OCL formulae is defined in the context of a MOF meta-model and its dynamic behavior specified by action semantics using the M3Actions framework.

Since all these temporal extensions of OCL are based on some temporal logic and include temporal logic operators, they intrinsically inherit the limitations of other specification approaches based on temporal logic: 1) they require strong theoretical and mathematical background, which are rarely found among practitioners; 2) they provided limited tool support, often based on prototypes that do not scale for industrial applications.

A different type of support for temporal constraints is proposed by Cabot et al. [21]. They extend UML to use UML/OCL as a temporal conceptual modeling language, introducing the concepts of *durability* and *frequency* for the definition of temporal features of UML classifiers and associations. They define temporal operations in OCL through which it is possible to refer to any past state of the system. These operations are mapped into standard OCL by relying on the mapping of the temporally-extended conceptual schema into a conventional UML one, which explicitly instantiates the concepts of time interval and instant. However, the temporal operations are geared to express temporal integrity constraints on the model, rather than temporal properties correlating events of the system.

8.2 Trace Checking and Run-time Verification

Model-driven technologies have been used in various work on (run-time) trace and/or assertion checking. The model-driven approach for assertion checking proposed in [61] relies on the principles of aspect-oriented programming and uses a technique called two-level aspect weaving. First, cross-cutting assertions defined using ECL, an extension of OCL, are weaved into a model defined within GME (Generic Modeling Environment [24]) and then the code for checking the contracts specified in the models is generated using model-driven program transformations [37]. ECL does not support the expression of temporal constraints. An approach conceptually similar to ours is proposed in [30], in which pre- and post-conditions are expressed with visual contracts defined using graph transformations and then transformed into a code-level representation as JML (Java Modeling Language) assertions. The pre- and post-conditions that can be expressed in this framework are functional and do not support temporal expressions. Reference [58] proposes a model-driven approach for monitoring Web services in which temporal properties, expressed using property specification patterns [27], are defined with a subset of UML 2.0 Sequence Diagrams and checked at run time by translating sequence diagrams into non-deterministic finite automata. However, the properties used in this work, differently from those that can be expressed with *TemPsy*, do not support expressing timing requirements. Our model-driven approach for trace checking can be easily applied in scenarios where other trace models are used, as long as OCL invariants can be expressed on them; examples of these models are those proposed in [20] (designed for the reverse engineering of UML sequence diagrams from traces) and [40] (tailored for the exchange of traces corresponding to large program call trees).

This work is also related to the more general area of run-time verification [47]. The majority of the approaches proposed in this area (e.g., [6,10,11,32], including previous work of some of the authors [12,14]) focuses on the verification of temporal properties expressed using some temporal logic. These approaches define the trace checking/run-time verification problem in terms of a *word problem*, i.e., the problem of whether a given word is included in some languages, and rely on formal verification tools like model checkers or SAT/SMT solvers. In our approach, we use a domain-specific specification language (*TemPsy*) and rely on standard MDE technologies.

9 Conclusion and Future Work

Offline trace checking is a procedure for checking the compliance of a system with respect to its requirements, by analyzing the log of events produced by the system at run time. We are interested in the offline trace checking of business processes and apply it, as a case study, to the particular context of eGovernment, in collaboration with our public service partner CTIE.

The goal of this paper is to present a practical and scalable solution for the offline checking of the temporal requirements of business processes, which can be used in contexts where model-driven engineering is already a practice, where temporal specifications should be written in a domain-specific language not requiring a strong mathematical background, and where relying on standards and industry-strength tools for property checking is a fundamental prerequisite.

This paper has made the following contributions: 1) the *TemPsy* language, a domain-specific specification language based on common property specification patterns and extended with new constructs, to facilitate the specification of business process requirements to be checked on traces; 2) a model-driven trace checking procedure, which relies on the efficient mapping of temporal requirements written in *TemPsy* into OCL constraints on a conceptual model of execution traces, which can be evaluated using an OCL checker; 3) the implementation of this trace checking procedure in the TemPsy-Check tool, which has been made publicly available; 4) the evaluation of the scalability of TemPsy-Check, applied to the verification of real properties derived from a case study of our public service partner, including a comparison with a state-of-the-art alternative technology based on temporal logic.

The results of the evaluation show the feasibility of applying our model-driven approach for offline trace checking in realistic settings. TemPsy-Check scales linearly with respect to the length of the input trace to check and is able to analyze traces with one million events in about two seconds. Moreover, it compares favorably with the state-of-the-art.

This work is part of a broader project in collaboration with CTIE, on model-driven run-time verification of business processes [26]. The next step is to embed our trace checking approach in the business process execution platform of our partner, to realize an efficient run-time verification platform for temporal properties of business process-based applications.

In addition, as part of future work, we plan to conduct a usability study of *TemPsy*, to assess the improved usability with respect to other specification methods (e.g., temporal logic). We also plan to apply our model-driven trace checking approach in other contexts different from business process modeling, with the possibility of extending *TemPsy* with additional constructs, as needed by the new application domains.

Acknowledgments

This work has been supported by the National Research Fund, Luxembourg (FNR/P10/03). We would like to thank the members of the Prometa team at CTIE, in particular Ludwig Balmer, Manuel Rouard, and Mathieu Syben, for their help with the analysis of the case study.

References

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A pattern language. Towns, buildings, construction.* Oxford University Press, 1977.

[2] 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.

[3] L. Baresi, S. Guinea, M. Pistore, and M. Trainotti. Dynamo + astro: An integrated approach for BPEL monitoring. In *Proc. ICWS '09*, pages 230–237. IEEE, July 2009.

[4] Luciano Baresi, Domenico Bianculli, Carlo Ghezzi, Sam Guinea, and Paola Spoletini. Validation of web service compositions. *IET Softw.*, 1(6):219–232, December 2007.

[5] Luciano Baresi and Sam Guinea. Towards dynamic monitoring of WS-BPEL processes. In *Proc. ICSOC 2005*, volume 3826 of *LNCS*, pages 269–282. Springer, 2005.

[6] Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé. MapReduce for parallel trace validation of LTL properties. In *Proc. RV 2012*, volume 7687 of *LNCS*, pages 184–198. Springer, 2013.

[7] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *Proc. FM 2012*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.

[8] 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, 2014.

[9] 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, 2012.

[10] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. In *Proc. RV 2013*, volume 8174 of *LNCS*, pages 40–58. Springer, 2013.

[11] David Basin, Felix Klaedtke, Samuel Müller, and Birgit Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proc. FSTTCS '08*, pages 49–60. IBFI Schloss Dagstuhl, 2008.

[12] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. SMT-based checking of SOLOIST over sparse traces. In *Proc. FASE 2014*, volume 8411 of *LNCS*, pages 276–290. Springer, April 2014.

[13] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Efficient large-scale trace checking using MapReduce. In *Proc. ICSE 2016*. ACM, May 2016. to be published.

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

[15] 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. IEEE, 2012.

[16] 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. Springer, 2013.

[17] Domenico Bianculli, Carlo Ghezzi, and Paola Spoletini. A model checking approach to verify BPEL4WS workflows. In *Proc. SOCA '07*, pages 13–20. IEEE, June 2007.

[18] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. Model checking of CTL-extended OCL specifications. In *Proc. SLE 2014*, volume 8706 of *LNCS*, pages 221–240. Springer, 2014.

[19] Marco Brambilla, Stefano Butti, and Piero Fraternali. WebRatio BPM: A tool for designing and deploying business processes on the web. In *Proc. ICWE 2010*, volume 6189 of *LNCS*, pages 415–429. Springer, 2010.

[20] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Trans. Softw. Eng.*, 32(9):642–663, September 2006.

[21] Jordi Cabot, Antoni Olivé, and Ernest Teniente. Representing temporal information in UML. In *Proc. UML 2003*, volume 2863 of *LNCS*, pages 44–59. Springer, 2003.

[22] MaríaVictoria Cengarle and Alexander Knapp. Towards OCL/RT. In *Proc. FME 2002*, volume 2391 of *LNCS*, pages 390–409. Springer, 2002.

[23] Stefan Conrad and Klaus Turowski. Temporal OCL: Meeting specification demands for business components. In *Unified Modeling Language*, pages 151–165. IGI Global, 2001.

[24] James Davis. GME: The generic modeling environment. In *Companion of the Proc. of OOPSLA '03*, pages 82–83. ACM, 2003.

[25] Wei Dou, Domenico Bianculli, and Lionel Briand. OCLR: a more expressive, pattern-based temporal extension of OCL. In *Proc. ECMFA 2014*, volume 8569 of *LNCS*, pages 51–66. Springer, July 2014.

[26] 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. Springer, September 2014.

[27] 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. IEEE, 1999.

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

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

[30] Gregor Engels, Marc Lohmann, Stefan Sauer, and Reiko Heckel. Model-driven monitoring: An application of graph transformation for design by contract. In *Proc. ICGT 2006*, volume 4178 of *LNCS*, pages 336–350. Springer, 2006.

[31] Miguel Felder and Angelo Morzenti. Validating real-time systems by history-checking TRIO specifications. *ACM Trans. Softw. Eng. Methodol.*, 3(4):308–339, October 1994.

[32] Bernd Finkbeiner, Sriram Sankaranarayanan, and HennyB. Sipma. Collecting statistics over runtime executions. *Form. Method Syst. Des.*, 27:253–274, 2005.

[33] Stephan Flake and Wolfgang Mueller. Past- and future-oriented time-bounded temporal properties with OCL. In *Proc. SEFM 2004*, pages 154–163. IEEE, 2004.

[34] Stephan Flake and Wolfgang Müller. Expressing property specification patterns with OCL. In *Proc. SERP '03*, pages 595–603. CSREA Press, 2003.

[35] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *Proc. WWW '04*, pages 621–630. ACM, 2004.

[36] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[37] Jeff Gray, Jing Zhang, Yuehua Lin, Suman Roychoudhury, Hui Wu, Rajesh Sudarsan, Aniruddha Gokhale, Sandeep Neema, Feng Shi, and Ted Bapty. Model-driven program transformation of a large avionics framework. In *Proc. GPCE 2004*, volume 3286 of *LNCS*, pages 361–378. Springer, 2004.

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

[39] Lars Grunske. Specification patterns for probabilistic quality properties. In *Proc. ICSE 2008*, pages 31–40. ACM, 2008.

[40] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A metamodel for the compact but lossless exchange of execution traces. *Softw. Syst. Model.*, 11(1):77–98, February 2012.

[41] Slim Kallel, Anis Charfi, Tom Dinkelaker, Mira Mezini, and Mohamed Jmaiel. Specifying and monitoring temporal properties in web services compositions. In *Proc. ECOWS '09*, pages 148–157. IEEE Computer Society, 2009.

[42] Bilal Kanso and Safouan Taha. Specification of temporal properties with OCL. *Sci. Comput. Program.*, 96, Part 4:527–551, 2014.

[43] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proc. ICSE '05*, pages 372–381. ACM, 2005.

[44] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, November 1990.

[45] Juliana Küster-Filipe and Stuart Anderson. On a time enriched OCL liveness template. *STTT*, 8(2):156–166, 2006.

[46] Luigi Lavazza, Sandro Morasca, and Angelo Morzenti. A dual language approach extension to UML for the development of time-critical component-based systems. *Electron. Notes Theor. Comput. Sci.*, 82(6):121–132, 2003.

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

[48] Zheng Li, Jun Han, and Yan Jin. Pattern-based specification and validation of web services interaction properties. In *Proc. ICSOC 2005*, volume 3826 of *LNCS*, pages 73–86. Springer, 2005.

[49] Markus Lumpe, Indika Meedeniya, and Lars Grunske. PSPWizard: machine-assisted definition of temporal logical properties with specification patterns. In *Proc. ESEC/FSE '11*, pages 468–471. ACM, 2011.

[50] Angelo Morzenti, Dino Mandrioli, and Carlo Ghezzi. A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14:521–573, October 1992.

[51] Aouatef Mrad, Samatar Ahmed, Sylvain Hallé, and Èric Beaudet. BabelTrace: A collection of transducers for trace validation. In *Proc. RV 2012*, volume 7687 of *LNCS*, pages 126–130. Springer, 2013.

[52] Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. RiTHM: A tool for enabling time-triggered runtime verification for C programs. In *Proc. ESEC/FSE 2013*, pages 603–606. ACM, 2013.

[53] OMG. BPMN Specification. `http://www.bpmn.org`, January 2011.

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

[55] Amalinda Post, Igor Menzel, Jochen Hoenicke, and Andreas Podelski. Automotive behavioral requirements expressed in a specification pattern system: A case study at BOSCH. *Requir. Eng.*, 17(1):19–33, March 2012.

[56] Franco Raimondi, James Skene, and Wolfgang Emmerich. Efficient online monitoring of web-service SLAs. In *Proc. SIGSOFT '08/FSE-16*, pages 170–180. ACM, 2008.

[57] William N. Robinson. Extended OCL for goal monitoring. *ECEASST*, 9, 2008.

[58] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse. Runtime monitoring of web service conversations. *IEEE Trans. Serv. Comput.*, 2(3):223–244, 2009.

[59] Michael Soden and Hajo Eichler. Temporal extensions of OCL revisited. In *Proc. ECMDA-FA*, volume 5562 of *LNCS*, pages 190–205. Springer, 2009.

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

[61] Jing Zhang, Jeff Gray, and Yuehua Lin. A model-driven approach to enforce crosscutting assertion checking. In *Proc. MACS '05*, pages 1–5. ACM, 2005.

[62] Paul Ziemann and Martin Gogolla. OCL extended with temporal logic. In *Proc. PSI 2003*, volume 2890 of *LNCS*, pages 351–357. Springer, 2003.

Appendix A OCL definitions

In this section we present the full definition of the OCL functions sketched in section 6. For implementation reasons, they have been defined in the context of the `Monitor` class.

A.1 Auxiliary Operations

functions invoked when applying scopes and checking patterns

```
context Monitor

=======
def: ordinalIndexOf(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer):Integer =
//find the index of the n-th occurrence of the event 'eventName'
let result:Tuple(index:Integer, ordinal:Integer) = trace->iterate(elem:trace::TraceElement;
  iter:Tuple(index:Integer, ordinal:Integer) = Tuple{index:Integer = 0, ordinal:Integer = 0}
  |
  if iter.ordinal = n then
    iter
  else
    if elem.event = eventName then
      Tuple{index:Integer = iter.index + 1, ordinal:Integer = iter.ordinal + 1}
    else
      Tuple{index:Integer = iter.index + 1, ordinal:Integer = iter.ordinal}
    endif
  endif
)
in
if result.ordinal = n then
  result.index
else
  -1
endif
=======
def: compare(a:Integer, b:Integer, which:Integer):Boolean =
if which = 1 then // at least b tu
  a >= b
else
  if which = 2 then // at most b tu
    a <= b
  else
    if which = 3 then // exactly b tu
      a = b
    else
      true // no comparison is needed
    endif
  endif
endif

=======
def: loadDistances(distances:Sequence(TemPsy::TimeDistance))
                  :Sequence(Tuple(which:Integer, value:Integer)) =
if distances->forAll(elem | elem->isEmpty()) then
  Sequence{}
else
  distances->iterate(elem:TemPsy::TimeDistance;
    iter:Sequence(Tuple(which:Integer, value:Integer)) = Sequence{}
    |
    if elem->isEmpty() then
      iter->append(Tuple{which:Integer=0, value:Integer=1})
    else
      if TemPsy::ComparingOperator::ATLEAST = elem.comparingOperator then
        iter->append(Tuple{which:Integer=1, value:Integer=elem.value})
      else
        if TemPsy::ComparingOperator::ATMOST = elem.comparingOperator then
          iter->append(Tuple{which:Integer=2, value:Integer=elem.value})
        else
          iter->append(Tuple{which:Integer=3, value:Integer=elem.value})
        endif
```

```
      endif
    endif)
endif
```

A.2 Scopes

functions for selecting segment(s) from the input trace, according to a scope definition

```
context Monitor

=======
def: applyScopeGlobally(trace:trace::Trace,
                        scope:TemPsy::Scope):OrderedSet(trace::TraceElement) =
trace.traceElements

=======
def: applyScopeBefore(trace:trace::Trace, scope:TemPsy::Scope):OrderedSet(trace::TraceElement) =
//return the scope of 'before boundary'
//'boundary' : '[n] eventName [comparingOperator timeDistance tu]'
let boundary:TemPsy::Boundary = scope.oclAsType(TemPsy::UniScope).boundary, eventName:String = boundary.event.name in
if boundary.timeDistance->notEmpty() then
  let comparingOperator:TemPsy::ComparingOperator = boundary.timeDistance.comparingOperator, timeDistance:Integer =
      boundary.timeDistance.value in
  if boundary.ordinal > 0 then
    let n:Integer = boundary.ordinal in
    if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
      self.atLeastBefore(trace.traceElements, eventName,n,timeDistance)
    else
      if TemPsy::ComparingOperator::ATMOST = comparingOperator then
        self.atMostBefore(trace.traceElements, eventName,n,timeDistance)
      else
        self.exactlyBefore(trace.traceElements, eventName,n,timeDistance)
      endif
    endif
  else
    if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
      self.atLeastBefore(trace.traceElements, eventName,1,timeDistance)
    else
      if TemPsy::ComparingOperator::ATMOST = comparingOperator then
        self.atMostBefore(trace.traceElements, eventName,1,timeDistance)
      else
        self.exactlyBefore(trace.traceElements, eventName,1,timeDistance)
      endif
    endif
  endif
else
  if boundary.ordinal > 0 then
    let n:Integer = boundary.ordinal in
    self.atLeastBefore(trace.traceElements, eventName,n,1)
  else
    self.atLeastBefore(trace.traceElements, eventName,1,1)
  endif
endif

=======
def: atLeastBefore(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):
     OrderedSet(trace::TraceElement) =
//return the scope of 'before [n] eventName at least timeDistance tu'
let position:Integer = ordinalIndexOf(trace, eventName, n) in
if 1 <> position.abs() then
  if 1 = timeDistance then
    trace->subOrderedSet(1, position-1)
  else
    let toTimeStamp:Integer = trace->at(position).timestamp in
    trace->select(elem | toTimeStamp - timeDistance >= elem.timestamp)
  endif
else
  OrderedSet{}
endif
```

```
=======
def: atMostBefore(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer): OrderedSet(trace::TraceElement) =
//return the scope of 'before [n] eventName at most timeDistance tu'
let position:Integer = ordinalIndexOf(trace, eventName, n) in
if -1 <> position then
  let toTimeStamp:Integer = trace->at(position).timestamp in
  trace->select(elem | toTimeStamp - timeDistance <= elem.timestamp and toTimeStamp >= elem.timestamp)
else
  OrderedSet{}
endif

=======
def: exactlyBefore(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer): OrderedSet(trace::TraceElement) =
//return the scope of 'before [n] eventName exactly timeDistance tu'
let position:Integer = ordinalIndexOf(trace, eventName, n) in
if -1 <> position then
  let toTimeStamp:Integer = trace->at(position).timestamp in
  trace->select(elem | toTimeStamp - timeDistance = elem.timestamp)
else
  OrderedSet{}
endif

=======
def: applyScopeAfter(trace:trace::Trace, scope:TemPsy::Scope):OrderedSet(trace::TraceElement) =
//return the scope of 'after boundary'
//'boundary' : '[n] eventName [comparingOperator timeDistance tu]'
let boundary:TemPsy::Boundary = scope.oclAsType(TemPsy::UniScope).boundary, eventName:String = boundary.event.name in
if boundary.timeDistance->notEmpty() then
  let comparingOperator:TemPsy::ComparingOperator = boundary.timeDistance.comparingOperator, timeDistance:Integer = boundary.timeDistance.value in
  if boundary.ordinal > 0 then
      let n:Integer = boundary.ordinal in
    if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
      self.atLeastAfter(trace.traceElements, eventName, n, timeDistance)
    else if TemPsy::ComparingOperator::ATMOST = comparingOperator then
        self.atMostAfter(trace.traceElements, eventName,n,timeDistance)
      else
        self.exactlyAfter(trace.traceElements, eventName, n, timeDistance)
      endif
    endif
  else
    if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
      self.atLeastAfter(trace.traceElements, eventName, 1,timeDistance)
    else if TemPsy::ComparingOperator::ATMOST = comparingOperator then
        self.atMostAfter(trace.traceElements, eventName, 1, timeDistance)
      else
        self.exactlyAfter(trace.traceElements, eventName, 1, timeDistance)
      endif
    endif
  endif
else
  if boundary.ordinal > 0 then
      let n:Integer = boundary.ordinal in
    self.atLeastAfter(trace.traceElements, eventName, n, 1)
    else
    self.atLeastAfter(trace.traceElements, eventName, 1, 1)
  endif
endif

=======
def: atLeastAfter(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer): OrderedSet(trace::TraceElement) =
//return the scope of 'after [n] eventName at least timeDistance tu'
let position:Integer = ordinalIndexOf(trace, eventName, n), size:Integer = trace->size() in
if -1 <> position and size <> position then
  if 1 = timeDistance then
```

```
    trace->subOrderedSet(position+1, size)
  else
    let fromTimeStamp:Integer = trace->at(position).timestamp in
    trace->select(elem | fromTimeStamp + timeDistance <= elem.timestamp)
  endif
else
  OrderedSet{}
endif

=======
def: atMostAfter(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):OrderedSet(trace::TraceElement) =
//return the scope of 'after [n] eventName at most timeDistance tu'
let position:Integer = ordinalIndexOf(trace, eventName, n) in
if -1 <> position then
  let fromTimeStamp:Integer = trace->at(position).timestamp in
  trace->select(elem | fromTimeStamp <= elem.timestamp and fromTimeStamp + timeDistance >= elem.timestamp)
else
  OrderedSet{}
endif

=======
def: exactlyAfter(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):OrderedSet(trace::TraceElement) =
//return the scope of 'after [n] eventName exactly timeDistance tu'
let position:Integer = ordinalIndexOf(trace, eventName, n) in
if -1 <> position then
  let fromTimeStamp:Integer = trace->at(position).timestamp in
  trace->select(elem | fromTimeStamp + timeDistance = elem.timestamp)
else
  OrderedSet{}
endif

=======

def: applyScopeBetweenAnd(trace:trace::Trace,
                          scope:TemPsy::Scope)
                         :OrderedSet(OrderedSet(trace::TraceElement)) =
// return the scope of 'between boundaryBegin and boundaryEnd'
// i.e., 'between [nBegin] eventNameBegin [at least timeDistanceBegin]
// and [nEnd] eventNameEnd [at least timeDistanceEnd]'
let boundaryBegin:TemPsy::Boundary
    = scope.oclAsType(TemPsy::BiScope).boundaryBegin,
    boundaryEnd:TemPsy::Boundary
    = scope.oclAsType(TemPsy::BiScope).boundaryEnd,
    eventNameBegin:String
    = boundaryBegin.event.name,
    eventNameEnd:String
    = boundaryEnd.event.name
in
if boundaryBegin.timeDistance->notEmpty() then
  let timeDistanceBegin:Integer = boundaryBegin.timeDistance.value in
  if boundaryEnd.timeDistance->notEmpty() then
    let timeDistanceEnd:Integer = boundaryEnd.timeDistance.value in
    if boundaryBegin.ordinal > 0 then
      let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
          nBegin:Integer = boundaryBegin.ordinal
      in
      if boundaryEnd.ordinal > 0 then
        let nEnd:Integer = boundaryEnd.ordinal in
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, nBegin, timeDistanceBegin,
                                      eventNameEnd, nEnd, timeDistanceEnd))
      else
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, nBegin, timeDistanceBegin,
                                      eventNameEnd, 1, timeDistanceEnd))
```

```
      endif
    else
      if boundaryEnd.ordinal > 0 then
        let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
            nEnd:Integer = boundaryEnd.ordinal
        in
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, 1, timeDistanceBegin,
                                      eventNameEnd, nEnd, timeDistanceEnd))
      else
        self.applyOriginalBetweenAnd(trace.traceElements,
                                    eventNameBegin, timeDistanceBegin,
                                    eventNameEnd, timeDistanceEnd)
      endif
    endif
  else
    if boundaryBegin.ordinal > 0 then
      let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
          nBegin:Integer = boundaryBegin.ordinal
      in
      if boundaryEnd.ordinal > 0 then
        let nEnd:Integer = boundaryEnd.ordinal in
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, nBegin, timeDistanceBegin,
                                      eventNameEnd, nEnd, 1))
      else
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, nBegin, timeDistanceBegin,
                                      eventNameEnd, 1, 1))
      endif
    else
      if boundaryEnd.ordinal > 0 then
        let result:OrderedSet(OrderedSet(trace::TraceElement))
            = OrderedSet{},
            nEnd:Integer = boundaryEnd.ordinal
        in
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, 1, timeDistanceBegin,
                                      eventNameEnd, nEnd, 1))
      else
        self.applyOriginalBetweenAnd(trace.traceElements,
                                    eventNameBegin, timeDistanceBegin, eventNameEnd)
      endif
    endif
  endif
else
  if boundaryEnd.timeDistance->notEmpty() then
    let timeDistanceEnd:Integer = boundaryEnd.timeDistance.value in
    if boundaryBegin.ordinal > 0 then
      let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
          nBegin:Integer = boundaryBegin.ordinal
      in
      if boundaryEnd.ordinal > 0 then
        let nEnd:Integer = boundaryEnd.ordinal in
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, nBegin, 1, eventNameEnd,
                                      nEnd, timeDistanceEnd))
      else
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, nBegin, 1,
                                      eventNameEnd, 1, timeDistanceEnd))
      endif
    else
```

```
      if boundaryEnd.ordinal > 0 then
        let result:OrderedSet(OrderedSet(trace::TraceElement))
            = OrderedSet{},
            nEnd:Integer = boundaryEnd.ordinal
        in
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, 1, 1,
                                      eventNameEnd, nEnd, timeDistanceEnd))
      else
        self.applyOriginalBetweenAnd(trace.traceElements,
                                     eventNameBegin, eventNameEnd, timeDistanceEnd)
      endif
    endif
  else
    if boundaryBegin.ordinal > 0 then
      let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
          nBegin:Integer = boundaryBegin.ordinal
      in
      if boundaryEnd.ordinal > 0 then
        let nEnd:Integer = boundaryEnd.ordinal in
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, nBegin, 1,
                                      eventNameEnd, nEnd, 1))
      else
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, nBegin, 1,
                                      eventNameEnd, 1, 1))
      endif
    else
      if boundaryEnd.ordinal > 0 then
        let result:OrderedSet(OrderedSet(trace::TraceElement))
            = OrderedSet{},
            nEnd:Integer = boundaryEnd.ordinal
        in
        result->append(
          self.applySpecialBetweenAnd(trace.traceElements,
                                      eventNameBegin, 1, 1,
                                      eventNameEnd, nEnd, 1))
      else
        self.applyOriginalBetweenAnd(trace.traceElements, eventNameBegin, eventNameEnd)
      endif
    endif
  endif
endif

=======
def: applyOriginalBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, eventNameEnd:String):
     Sequence(OrderedSet(trace::TraceElement)) =
//return the scope of 'between eventNameBegin and eventNameEnd'
trace->iterate(elem:trace::TraceElement;
  iter:Tuple(index:Integer, result:Sequence(OrderedSet(trace::TraceElement)), i:Integer)
  =Tuple{index:Integer = 0, result:Sequence(OrderedSet(trace::TraceElement)) = Sequence{}, i:Integer = 0} |
  if iter.i = 0 then
    let currentIndex:Integer = iter.index + 1 in
    if elem.event = eventNameBegin then
      Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
            currentIndex}
    else
      Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
            iter.i}
    endif
  else
    if elem.event = eventNameEnd then
      let i:Integer = iter.i+1, j:Integer = iter.index in
      if i <= j then
        Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result->append(trace->
```

```
        subOrderedSet(i, j)), i:Integer = 0}
    else
      Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = 0}
    endif
  else
    Tuple{index:Integer = iter.index + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = iter.i}
  endif
endif
).result

=======
def: applyOriginalBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, distanceBegin:Integer, eventNameEnd:String):Sequence(OrderedSet(trace::TraceElement)) =
//return the scope of 'between eventNameBegin at least distanceBegin tu and eventNameEnd'
trace->iterate(elem:trace::TraceElement;
  iter:Tuple(index:Integer, result:Sequence(OrderedSet(trace::TraceElement)), i:Integer, criticalTime:Integer)
  =Tuple{index:Integer = 0, result:Sequence(OrderedSet(trace::TraceElement)) = Sequence{}, i:Integer = 0, criticalTime:Integer = 0} |
  let e:String = elem.event in
  if iter.i = 0 then
    let currentIndex:Integer = iter.index + 1 in
    if e = eventNameBegin then
      Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = currentIndex, criticalTime:Integer = elem.timestamp + distanceBegin}
    else
      Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = iter.i, criticalTime:Integer = iter.criticalTime}
    endif
  else
    if e = eventNameEnd then
      let t:Integer = elem.timestamp, i:Integer = iter.i + 1, j:Integer = iter.index, t1:Integer = iter.criticalTime in
      if i <= j and t1 < t then
        Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result->append(trace->subOrderedSet(i, j)->select(segElem | segElem.timestamp >= t1)), i:Integer = 0, criticalTime:Integer = iter.criticalTime}
      else
        Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = 0, criticalTime:Integer = iter.criticalTime}
      endif
    else
      Tuple{index:Integer = iter.index + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = iter.i, criticalTime:Integer = iter.criticalTime}
    endif
  endif
).result

=======
def: applyOriginalBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, eventNameEnd:String, distanceEnd:Integer):Sequence(OrderedSet(trace::TraceElement)) =
//return the scope of 'between eventNameBegin and at least distanceEnd tu eventNameEnd'
trace->iterate(elem:trace::TraceElement;
  iter:Tuple(index:Integer, result:Sequence(OrderedSet(trace::TraceElement)), i:Integer, criticalTime:Integer)
  =Tuple{index:Integer = 0, result:Sequence(OrderedSet(trace::TraceElement)) = Sequence{}, i:Integer = 0, criticalTime:Integer = 0} |
  let e:String = elem.event in
  if iter.i = 0 then
    let currentIndex:Integer = iter.index + 1 in
    if e = eventNameBegin then
      Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = currentIndex, criticalTime:Integer = elem.timestamp + 1}
    else
      Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = iter.i, criticalTime:Integer = iter.criticalTime}
    endif
  else
    if e = eventNameEnd then
      let t:Integer = elem.timestamp, i:Integer = iter.i + 1, j:Integer = iter.index, t1:Integer = iter.criticalTime,
```

```
      t2:Integer = t - distanceEnd in
    if i <= j and t1 <= t2 then
      Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result->append(trace->subOrderedSet(i, j)->select(segElem | segElem.timestamp <= t2)), i:Integer = 0, criticalTime:Integer = iter.criticalTime}
    else
      Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = 0, criticalTime:Integer = iter.criticalTime}
    endif
  else
    Tuple{index:Integer = iter.index + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = iter.i, criticalTime:Integer = iter.criticalTime}
  endif
endif
).result

=======
def: applyOriginalBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, distanceBegin:Integer, eventNameEnd:String, distanceEnd:Integer):Sequence(OrderedSet(trace::TraceElement)) =
//return the scope of 'between eventNameBegin at least distanceBegin tu and at least distanceEnd tu eventNameEnd'
trace->iterate(elem:trace::TraceElement;
  iter:Tuple(index:Integer, result:Sequence(OrderedSet(trace::TraceElement)), i:Integer, criticalTime:Integer)
  =Tuple{index:Integer = 0, result:Sequence(OrderedSet(trace::TraceElement)) = Sequence{}, i:Integer = 0, criticalTime:Integer = 0} |
  let e:String = elem.event in
  if iter.i = 0 then
    let currentIndex:Integer = iter.index + 1 in
    if e = eventNameBegin then
      Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = currentIndex, criticalTime:Integer = elem.timestamp + distanceBegin}
    else
      Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = iter.i, criticalTime:Integer = iter.criticalTime}
    endif
  else
    if e = eventNameEnd then
      let t:Integer = elem.timestamp, i:Integer = iter.i + 1, j:Integer = iter.index, t1:Integer = iter.criticalTime, t2:Integer = t - distanceEnd in
      if i <= j and t1 <= t2 then
        Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result->append(trace->subOrderedSet(i, j)->select(segElem | segElem.timestamp >= t1 and segElem.timestamp <= t2)), i:Integer = 0, criticalTime:Integer = iter.criticalTime}
      else
        Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = 0, criticalTime:Integer = iter.criticalTime}
      endif
    else
      Tuple{index:Integer = iter.index + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = iter.i, criticalTime:Integer = iter.criticalTime}
    endif
  endif
).result

=======
def: applySpecialBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, nBegin:Integer, timeDistanceBegin:Integer, eventNameEnd:String, nEnd:Integer, timeDistanceEnd:Integer):OrderedSet(trace::TraceElement) =
//return the scope of 'between nBegin eventNameBegin at least timeDistanceBegin tu and nBegin eventNameEnd at least timeDistanceEnd tu'
let t:Tuple(index:Integer, indexBegin:Integer, indexEnd:Integer, count:Integer) = trace->iterate(elem:trace::TraceElement;
    iter:Tuple(index:Integer, indexBegin:Integer, indexEnd:Integer, count:Integer) = Tuple{index:Integer = 0, indexBegin:Integer = 0, indexEnd:Integer = 0, count:Integer = 0} |
    if iter.indexBegin = 0 then
      let currentIndex:Integer = iter.index + 1 in
      if elem.event = eventNameBegin then
        let currentBeginCount:Integer = iter.count+1 in
        if currentBeginCount = nBegin then
          Tuple{index:Integer = currentIndex, indexBegin:Integer = currentIndex + 1, indexEnd:Integer = iter.indexEnd
```

```
              , count:Integer = 0}
        else
          Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = iter.indexEnd,
               count:Integer = currentBeginCount}
        endif
      else
        Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = iter.indexEnd,
             count:Integer = iter.count}
      endif
    else
      if iter.indexEnd = 0 then
        let currentIndex:Integer = iter.index + 1 in
        if elem.event = eventNameEnd then
          let currentEndCount:Integer = iter.count+1 in
          if currentEndCount = nEnd then
            Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = currentIndex
                 -1, count:Integer = nEnd}
          else
            Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = iter.
                 indexEnd, count:Integer = currentEndCount}
          endif
        else
          Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = iter.indexEnd,
               count:Integer = iter.count}
        endif
      else
        iter
      endif
    endif
  )
in
let
  i:Integer = t.indexBegin,
  j:Integer = t.indexEnd,
  timestampBegin:Integer = trace->at(i-1).timestamp+timeDistanceBegin,
  timestampEnd:Integer = trace->at(j+1).timestamp-timeDistanceEnd
in
if i > 0 and j > 0 and i <= j then
  if timeDistanceBegin = 1 and timeDistanceEnd = 1 then
    trace->subOrderedSet(i, j)
  else
    trace->subOrderedSet(i, j)->select(elem | elem.timestamp >= timestampBegin and elem.timestamp <= timestampEnd)
  endif
else
  OrderedSet{}
endif
```

A.3 Patterns

functions for checking a given pattern on the trace segment(s) determined by a scope

```
context Monitor

=======
def: checkPatternUniversality(subtrace:OrderedSet(trace::TraceElement), pattern:TemPsy::Pattern):Boolean =
// check the satisfiability of the universality pattern 'always eventName'
let eventName:String = pattern.oclAsType(TemPsy::Universality).event.name in
subtrace->forAll(event = eventName)

=======
def: checkPatternExistence(subtrace:OrderedSet(trace::TraceElement), pattern:TemPsy::Pattern):Boolean =
--check the satisfiability of the existence pattern 'pattern'
if subtrace->isEmpty() then
  true
else
  let occPattern:TemPsy::OccurrencePattern = pattern.oclAsType(TemPsy::OccurrencePattern), eventName:String =
       occPattern.event.name in
  if occPattern.comparingOperator->notEmpty() then
```

```
    let comparingOperator:TemPsy::ComparingOperator = occPattern.comparingOperator, n:Integer = occPattern.times,
         count:Integer = subtrace.event->count(eventName) in
    if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
      count >= n
    else
      if TemPsy::ComparingOperator::ATMOST = comparingOperator then
        count <= n
      else
        count = n
      endif
    endif
  else
    subtrace.event->includes(eventName)
  endif
endif

=======
def: checkPatternAbsence(subtrace:OrderedSet(trace::TraceElement), pattern:TemPsy::Pattern):Boolean =
--check the satisfiability of the absence pattern 'pattern'
if subtrace->isEmpty() then
  true
else
  let occPattern:TemPsy::OccurrencePattern = pattern.oclAsType(TemPsy::OccurrencePattern), eventName:String =
       occPattern.event.name in
  if occPattern.comparingOperator->notEmpty() then
    subtrace.event->count(eventName) <> occPattern.times
  else
    subtrace.event->excludes(eventName)
  endif
endif

=======
            else
              self.checkPatternPrecedenceOneManyRight(subtrace, cause,
                                                      effects, effectDistances)
            endif
        else
          self.checkPatternPrecedenceManyManyRight(subtrace, causes,
                                                   effects, effectDistances)
        endif
      else
        if causeSize = 1 then
          let cause:String=causes->first() in
            if effectSize = 2 then
              let effectDistance:Tuple(which:Integer, value:Integer)
                  =effectDistances->first()
              in
              self.checkPatternPrecedenceOneTwoMidRight(subtrace, cause,
                                                        orderPattern.timeDistance,
                                                        effects->first(),
                                                        effectDistance, effects->at(2))
            else
              let distance:Tuple(which:Integer, value:Integer)
                  =self.loadDistance(orderPattern.timeDistance)
              in
              self.checkPatternPrecedenceOneManyMidRight(subtrace, cause,
                                                         distance, effects,
                                                         effectDistances)
            endif
        else
          let distance:Tuple(which:Integer, value:Integer)
              =self.loadDistance(orderPattern.timeDistance)
          in
          self.checkPatternPrecedenceManyManyMidRight(subtrace, causes,
                                                      distance, effects, effectDistances)
        endif
      endif
    endif
  else
```

```
    if effectDistances->isEmpty() then
      if orderPattern.timeDistance->isEmpty() then
        if effectSize = 1 then
          let effect:String=effects->first() in
          if causeSize = 2 then
            let causeDistance:Tuple(which:Integer, value:Integer)
                =causeDistances->first()
            in
            self.checkPatternPrecedenceTwoOneLeft(subtrace, causes->first(),
                                                  causeDistance, causes->at(2), effect)
          else
            self.checkPatternPrecedenceManyOneLeft(subtrace, causes,
                                                   causeDistances, effect)
          endif
        else
          self.checkPatternPrecedenceManyManyLeft(subtrace, causes,
                                                  causeDistances, effects)
        endif
      else
        let distance:Tuple(which:Integer, value:Integer)
            =self.loadDistance(orderPattern.timeDistance)
        in
        if effectSize = 1 then
          let effect:String=effects->first() in
          self.checkPatternPrecedenceManyOneLeftMid(subtrace, causes,
                                                    causeDistances, distance, effect)
        else
          self.checkPatternPrecedenceManyManyLeftMid(subtrace, causes,
                                                     causeDistances, distance, effects)
        endif
      endif
    else
      if orderPattern.timeDistance->isEmpty() then
        self.checkPatternPrecedenceManyManyLeftRight(subtrace, causes, causeDistances,
                                                     effects, effectDistances)
      else
        let distance:Tuple(which:Integer, value:Integer)
            =self.loadDistance(orderPattern.timeDistance)
        in
        self.checkPatternPrecedenceManyManyLeftMidRight(subtrace, causes, causeDistances,
                                                        distance, effects, effectDistances)
      endif
    endif
  endif
endif

=======
def: checkPatternPrecedenceOneOnePlain(subtrace:OrderedSet(trace::TraceElement), cause:String, effect:String):Boolean
     =
//"cause preceding effect"
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, result:Integer) = Tuple{flag:Boolean = true,
    result:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then
      Tuple{flag:Boolean = false, result:Integer = -1}
    else
      if e = effect then
        Tuple{flag:Boolean = false, result:Integer = -2} // violation
      else
        iter
      endif
    endif
  else
    iter
  endif
).result >= -1
```

```
=======
def: checkPatternPrecedenceOneOneAtLeastMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effect:String):Boolean =
//"cause preceding at least distance tu effect"
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstant:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.midCriticalInstant = 0 and e = cause then //catch the first occurrence of cause
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance}
    else
      if e = effect then
        if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2} // violation
        else
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1}
        endif
      else
        iter
      endif
    endif
  else
    iter
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceOneOneAtMostMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effect:String):Boolean =
//"cause preceding at most distance tu effect"
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstant:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then //latest cause
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance}
    else
      if e = effect and elem.timestamp > iter.midCriticalInstant then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = null} // violation
      else
        iter
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceOneOneExactlyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effect:String):Boolean =
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer)) = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}} |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.timestamp+distance)}
    else
      if e = effect then
        let t:Integer = elem.timestamp in
        if iter.midCriticalInstants->includes(t) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t)}
        else
          Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null}
        endif
```

```
      else
        iter
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceOneOneMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:TemPsy::
     TimeDistance, effect:String):Boolean =
let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
if which = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceOneOneAtLeastMid(subtrace, cause, value, effect)
else
  if which = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceOneOneAtMostMid(subtrace, cause, value, effect)
  else
    self.checkPatternPrecedenceOneOneExactlyMid(subtrace, cause, value, effect)
  endif
endif

=======
def: checkPatternPrecedenceOneManyPlain(subtrace:OrderedSet(trace::TraceElement), cause:String, effects:Sequence(
     String)):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, result:Integer, i2:Integer) = Tuple{flag:Boolean
      = true, result:Integer = 0, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then //catch the first occurrence of cause
      Tuple{flag:Boolean = false, result:Integer = -1, i2:Integer = null}
    else
      if e = effects->at(iter.i2) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = false, result:Integer = -2, i2:Integer = null}
        else
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = iter.i2 + 1}
        endif
      else
        if e = firstEffect then
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = 2}
        else
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).result >= -1

=======
def: checkPatternPrecedenceOneManyAtLeastMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer
     , effects:Sequence(String)):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer) = Tuple{
     flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
```

```
    if iter.midCriticalInstant = 0 and e = cause then //catch the first occurrence of cause
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i2:Integer = 1}
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i2:Integer = null}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2 + 1}
        endif
      else
        if e = firstEffect then
          if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
          else
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i2:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceOneManyAtMostMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effects:Sequence(String)):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1} |
  let e:String = elem.event in
  if iter.flag then
    if e = cause then
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i2:Integer = 1}
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i2:Integer = null}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2 + 1}
        endif
      else
        if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceOneManyExactlyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effects:Sequence(String)):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
```

```
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i2:Integer) = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.timestamp+distance), i2:Integer = 1}
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i2:Integer = null}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:Integer = iter.i2 + 1}
        endif
      else
        if e = firstEffect then
          let t:Integer = elem.timestamp in
          if iter.midCriticalInstants->includes(t) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i2:Integer = 1}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:Integer = 2}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceOneManyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:TemPsy::TimeDistance, effects:Sequence(String)):Boolean =
let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
if which = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceOneManyAtLeastMid(subtrace, cause, value, effects)
else
  if which = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceOneManyAtMostMid(subtrace, cause, value, effects)
  else
    self.checkPatternPrecedenceOneManyExactlyMid(subtrace, cause, value, effects)
  endif
endif


=======
def: checkPatternPrecedenceOneManyRight(subtrace:OrderedSet(trace::TraceElement), cause:String, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, result:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, result:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then
      Tuple{flag:Boolean = false, result:Integer = -1, i2:Integer = null, effectCriticalInstant:Integer = null}
    else
```

```
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = false, result:Integer = -2, i2:Integer = null, effectCriticalInstant:Integer = null}
        else
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
        endif
      else
        if e = firstEffect then
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
        else
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    endif
  else
    iter
  endif
).result >= -1

=======
def: checkPatternPrecedenceOneManyAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.midCriticalInstant = 0 and e = cause then //catch the first occurrence of cause
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i2:Integer = null, effectCriticalInstant:Integer = null}
        else
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
        endif
      else
        if e = firstEffect then
          if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
          else
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i2:Integer = null, effectCriticalInstant:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    endif
  else
    iter
```

```
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceOneManyAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance: Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant: Integer = null}
        else
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
        endif
      else
        if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    endif
  else
    iter
  endif
).flag

// added on 18/08/2015
def: checkPatternPrecedenceOneManyExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance :Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i2: Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem. timestamp+midDistance), i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i2:Integer = null,
```

```
                effectCriticalInstant:Integer = null}
        else
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
        endif
      else
        if e = firstEffect then
          let t:Integer = elem.timestamp in
          if iter.midCriticalInstants->includes(t) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceOneManyMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:TemPsy::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceOneManyAtLeastMidRight(subtrace, cause, midValue, effects, effectDistances)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceOneManyAtMostMidRight(subtrace, cause, midValue, effects, effectDistances)
  else
    self.checkPatternPrecedenceOneManyExactlyMidRight(subtrace, cause, midValue, effects, effectDistances)
  endif
endif

=======
def: checkPatternPrecedenceManyOnePlain(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first()
in
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, result:Integer, i1:Integer) = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null}
      else
        Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1}
      endif
    else
      if e = firstCause then
        Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2}
      else
        if e = effect then
          Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null}
        else
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1}
        endif
```

```
      endif
    endif
  else
    iter
  endif
).result >= -1

=======
def: checkPatternPrecedenceManyOneAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    distance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:
      Integer = 0, i1:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1}
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 +
            1}
      endif
    else
      if iter.midCriticalInstant = 0 and e = firstCause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
      else
        if e = effect then
          if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null}
          else
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceManyOneAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    distance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:
      Integer = 0, i1:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1}
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 +
            1}
      endif
    else
      if e = firstCause then
```

```
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
    else
      if e = effect and elem.timestamp > iter.midCriticalInstant then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null}
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
      endif
    endif
  endif
else
  iter
endif
).flag

=======
def: checkPatternPrecedenceManyOneExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.timestamp+distance), i1:Integer = 1}
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = iter.i1 + 1}
      endif
    else
      if e = firstCause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 2}
      else
        if e = effect then
          let t:Integer = elem.timestamp in
          if iter.midCriticalInstants->includes(t) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i1:Integer = 1}
          else
            Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceManyOneMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:TemPsy::TimeDistance, effect:String):Boolean =
let value:Integer = distance.value, which:TemPsy::ComparingOperator=distance.comparingOperator in
if which = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceManyOneAtLeastMid(subtrace, causes, value, effect)
else
  if which = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceManyOneAtMostMid(subtrace, causes, value, effect)
  else
```

```
    self.checkPatternPrecedenceManyOneExactlyMid(subtrace, causes, value, effect)
  endif
endif

=======
def: checkPatternPrecedenceManyOneLeft(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, result:Integer, i1:Integer, causeCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null}
      else
        let i11:Integer = iter.i1 + 1 in
        Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value}
      endif
    else
      if e = firstCause then
        Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance}
      else
        if e = effect then
          Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer = null}
        else
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
        endif
      endif
    endif
  else
    iter
  endif
).result >= -1

=======
def: checkPatternPrecedenceManyOneLeftAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then//imply iter.midCriticalInstant = 0
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
      else
        let i11:Integer = iter.i1 + 1 in
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value}
      endif
```

```
    else
      if iter.midCriticalInstant = 0 and e = firstCause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
             causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance}
      else
        if e = effect then
          if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, causeCriticalInstant:
                 Integer = null}
          else
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:
                 Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
               causeCriticalInstant:Integer = iter.causeCriticalInstant}
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceManyOneLeftAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
     causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
         causeDistances->at(iter.i1).which) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
             causeCriticalInstant:Integer = iter.causeCriticalInstant}
      else
        let i11:Integer = iter.i1 + 1 in
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
             causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value}
      endif
    else
      if e = firstCause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
             causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance}
      else
        if e = effect and elem.timestamp > iter.midCriticalInstant then
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:
               Integer = null}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
               causeCriticalInstant:Integer = iter.causeCriticalInstant}
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceManyOneLeftExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
```

```
    causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, causeCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1,
      causeCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
        causeDistances->at(iter.i1).which) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem
            .timestamp+midDistance), i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
      else
        let i11:Integer = iter.i1 + 1 in
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer
            = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value}
      endif
    else
      if e = firstCause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer
            = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance}
      else
        if e = effect then
          let t:Integer = elem.timestamp in
          if iter.midCriticalInstants->includes(t) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(
                subElem | subElem > t), i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
          else
            Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null,
                causeCriticalInstant:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
              Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceManyOneLeftMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TemPsy::TimeDistance, effect:String):
    Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceManyOneLeftAtLeastMid(subtrace, causes, causeDistances, midValue, effect)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceManyOneLeftAtMostMid(subtrace, causes, causeDistances, midValue, effect)
  else
    self.checkPatternPrecedenceManyOneLeftExactlyMid(subtrace, causes, causeDistances, midValue, effect)
  endif
endif

=======
def: checkPatternPrecedenceManyManyPlain(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effects:
    Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
```

```
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, result:Integer, i1:Integer, i2:Integer) = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, i2:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, i2:Integer = null}
        else
          if e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = 1}
            endif
          endif
        endif
      else
        if e = firstCause then
          if e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = 1}
            endif
          endif
        else
          if e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = 1}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).result >= -1

=======
def: checkPatternPrecedenceManyManyAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:Integer, effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1} |
```

```
  if iter.flag then
    let e:String = elem.event in
    if iter.midCriticalInstant > 0 and elem.timestamp >= iter.midCriticalInstant then
      Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, i2:Integer = null} // satisfaction
    else
      if iter.i2 = effectSize and e = lastEffect then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, i2:Integer = null} // violation
      else
        if iter.i1 > 1 and e = causes->at(iter.i1) then
          if iter.i1 = causeSize then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1, i2:Integer = 1}
          else
            if e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1} // a potential violation to time distance
            else
              if e = firstEffect then
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 2} // a potential violation
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 1}
              endif
            endif
          endif
        else
          if iter.midCriticalInstant = 0 and e = firstCause then
            if e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = iter.i2 + 1}
            else
              if e = firstEffect then
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 2} // a potential violation
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 1}
              endif
            endif
          else
            if e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = iter.i2 + 1}
            else
              if e = firstEffect then
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2} // a potential violation
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1}
              endif
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceManyManyAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:Integer, effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
```

```
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1, i2:Integer = 1}
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 1}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 1}
            endif
          endif
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceManyManyExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
```

```
    distance:Integer, effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null, i2:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(
               elem.timestamp+distance), i1:Integer = 1, i2:Integer = 1}
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                 Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
                     select(subElem | subElem > t), i1:Integer = iter.i1 + 1, i2:Integer = 1}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                     Integer = iter.i1 + 1, i2:Integer = 2}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                   Integer = iter.i1 + 1, i2:Integer = 1}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                 Integer = 2, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
                     select(subElem | subElem > t), i1:Integer = 2, i2:Integer = 1}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                     Integer = 2, i2:Integer = 2}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                   Integer = 2, i2:Integer = 1}
            endif
          endif
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                 Integer = 1, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
```

```
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i1:Integer = 1, i2:Integer = 1}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, i2:Integer = 2}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, i2:Integer = 1}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceManyManyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:TemPsy::TimeDistance, effects:Sequence(String)):Boolean =
let value:Integer = distance.value, which:TemPsy::ComparingOperator=distance.comparingOperator in
if which = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceManyManyAtLeastMid(subtrace, causes, value, effects)
else
  if which = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceManyManyAtMostMid(subtrace, causes, value, effects)
  else
    self.checkPatternPrecedenceManyManyExactlyMid(subtrace, causes, value, effects)
  endif
endif

=======
def: checkPatternPrecedenceManyManyLeft(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, result:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer) = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null}
        else
          let i11:Integer = iter.i1 + 1 in
          if e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = 2}
            else
```

```
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = 1}
            endif
          endif
        endif
      else
        if e = firstCause then
          if e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
            endif
          endif
        else
          if e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).result >= -1

=======
def: checkPatternPrecedenceManyManyLeftAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.midCriticalInstant > 0 and elem.timestamp >= iter.midCriticalInstant then
      Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null} // satisfaction
    else
      if iter.i2 = effectSize and e = lastEffect then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null} // violation
      else
        if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
```

```
          if iter.i1 = causeSize then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
          else
            let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value in
            if e = effects->at(iter.i2) then // for instance {causes: [a,b,c], effects: [d,a,b]}, when i1 = 1, i2 = 2 or i1 = 2, i2 = 3. But it is not possible i1 equals to causeSize, since causes cannot be a sublist of effects.
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = iter.i2 + 1}
            else
              if e = firstEffect then
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2} // a potential violation
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1}
              endif
            endif
          endif
        else
          if iter.midCriticalInstant = 0 and e = firstCause then
            if e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
            else
              if e = firstEffect then
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2} // a potential violation
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
              endif
            endif
          else
            if e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
            else
              if e = firstEffect then
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2} // a potential violation
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
              endif
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceManyManyLeftAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
```

```
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
        else
          let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value in
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
            endif
          endif
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).flag
```

```

=======
def: checkPatternPrecedenceManyManyLeftExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, causeCriticalInstant:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.timestamp+midDistance), i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
        else
          let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value in
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
```

```
Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
            endif
          endif
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceManyManyLeftMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TemPsy::TimeDistance, effects:Sequence(String)):Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceManyManyLeftAtLeastMid(subtrace, causes, causeDistances, midValue, effects)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceManyManyLeftAtMostMid(subtrace, causes, causeDistances, midValue, effects)
  else
    self.checkPatternPrecedenceManyManyLeftExactlyMid(subtrace, causes, causeDistances, midValue, effects)
  endif
endif

=======
def: checkPatternPrecedenceManyManyRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, result:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
```

```
      Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).result >= -1

=======
def: checkPatternPrecedenceManyManyAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
```

```
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:
      Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.midCriticalInstant > 0 and elem.timestamp >= iter.midCriticalInstant then
      Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, i2:Integer = null,
          effectCriticalInstant:Integer = null} // satisfaction
    else
      if iter.i2 = effectSize and e = lastEffect then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, i2:Integer = null,
            effectCriticalInstant:Integer = null} // violation
      else
        if iter.i1 > 1 and e = causes->at(iter.i1) then
          if iter.i1 = causeSize then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer =
                1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          else
            if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
                effectDistances->at(iter.i2).which) then
              let i22:Integer = iter.i2 + 1 in
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter
                  .i1 + 1, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(
                  i22).value}
            else
              if e = firstEffect then
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
                    iter.i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp +
                    secondEffectDistance} // a potential violation
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
                    iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
              endif
            endif
          endif
        else
          if iter.midCriticalInstant = 0 and e = firstCause then
            if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
                effectDistances->at(iter.i2).which) then
              let i22:Integer = iter.i2 + 1 in
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                  i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
            else
              if e = firstEffect then
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                    i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a
                    potential violation
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                    i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
              endif
            endif
          else
            if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
                effectDistances->at(iter.i2).which) then
              let i22:Integer = iter.i2 + 1 in
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                  i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
            else
              if e = firstEffect then//midCriticalInstant is either 0 or midCriticalInstant > elem.timestamp
                Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                    i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a
```

```
potential violation
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      endif
    endif
  endif
  else
    iter
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceManyManyAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
```

```
              i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
def: checkPatternPrecedenceManyManyExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.timestamp+midDistance), i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = iter.i1 + 1, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
```

```
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i1:Integer = iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = iter.i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 2, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i1:Integer = 2, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 2, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 2, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              let t:Integer = elem.timestamp in
              if iter.midCriticalInstants->includes(t) then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(subElem | subElem > t), i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
              else
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
              endif
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      endif
    endif
  else
    iter
  endif
).flag

=======
```

```
def: checkPatternPrecedenceManyManyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), midDistance:TemPsy::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceManyManyAtLeastMidRight(subtrace, causes, midValue, effects, effectDistances)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceManyManyAtMostMidRight(subtrace, causes, midValue, effects, effectDistances)
  else
    self.checkPatternPrecedenceManyManyExactlyMidRight(subtrace, causes, midValue, effects, effectDistances)
  endif
endif

=======
def: checkPatternPrecedenceManyManyLeftRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, result:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
        else
          let i11:Integer = iter.i1 + 1 in
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
            else
              Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
```

```
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
        else
          if e = firstEffect then
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      else
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
        else
          if e = firstEffect then
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    endif
  endif
else
  iter
endif
).result >= -1

=======
def: checkPatternPrecedenceManyManyLeftAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.midCriticalInstant > 0 and elem.timestamp >= iter.midCriticalInstant then
      Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null} // satisfaction
    else
      if iter.i2 = effectSize and e = lastEffect then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null} // violation
      else
```

```
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value in
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a potential violation
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      else
        if iter.midCriticalInstant = 0 and e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a potential violation
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        else
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a potential violation
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      endif
    endif
  endif
else
```

```
    iter
  endif
).midCriticalInstant >= -1

=======
def: checkPatternPrecedenceManyManyLeftAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value in
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
            else
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          endif
        endif
      else
        if e = firstCause then
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
          else
            if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
```

```
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      else
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
        else
          if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    endif
  endif
 else
  iter
 endif
).flag

=======
def: checkPatternPrecedenceManyManyLeftExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, causeCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.timestamp+midDistance), i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value in
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
            let i22:Integer = iter.i2 + 1 in
```

```
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
          Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = i22,
          effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
    else
      if e = firstEffect then
        let t:Integer = elem.timestamp in
        if iter.midCriticalInstants->includes(t) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
              select(subElem | subElem > t), i1:Integer = i11, causeCriticalInstant:Integer =
              nextCauseCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.
              effectCriticalInstant}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
              Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2,
              effectCriticalInstant:Integer = t + secondEffectDistance}
        endif
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
            Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1,
            effectCriticalInstant:Integer = iter.effectCriticalInstant}
      endif
    endif
  endif
else
  if e = firstCause then
    if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
      let i22:Integer = iter.i2 + 1 in
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
          Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = i22,
          effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
    else
      if e = firstEffect then
        let t:Integer = elem.timestamp in
        if iter.midCriticalInstants->includes(t) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
              select(subElem | subElem > t), i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp +
              secondCauseDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
              Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer =
              2, effectCriticalInstant:Integer = t + secondEffectDistance}
        endif
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
            Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1,
            effectCriticalInstant:Integer = iter.effectCriticalInstant}
      endif
    endif
  else
    if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
      let i22:Integer = iter.i2 + 1 in
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
          Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22,
          effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
    else
      if e = firstEffect then
        let t:Integer = elem.timestamp in
        if iter.midCriticalInstants->includes(t) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
              select(subElem | subElem > t), i1:Integer = 1, causeCriticalInstant:Integer = iter.
              causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant
              }
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
              Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2,
              effectCriticalInstant:Integer = t + secondEffectDistance}
        endif
```

```
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    endif
  endif
else
  iter
endif
).flag

=======
def: checkPatternPrecedenceManyManyLeftMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TemPsy::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternPrecedenceManyManyLeftAtLeastMidRight(subtrace, causes, causeDistances, midValue, effects, effectDistances)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternPrecedenceManyManyLeftAtMostMidRight(subtrace, causes, causeDistances, midValue, effects, effectDistances)
  else
    self.checkPatternPrecedenceManyManyLeftExactlyMidRight(subtrace, causes, causeDistances, midValue, effects, effectDistances)
  endif
endif

=======
def: checkPatternResponse(subtrace:OrderedSet(trace::TraceElement), pattern:TemPsy::Pattern):Boolean =
--check the satisfiability of the response pattern 'effect responding cause'
--in the first event in the chain 'effect', it may contains time distance to the last event in the chain 'cause'
if subtrace->isEmpty() then
  true
else
  let orderPattern:TemPsy::OrderPattern = pattern.oclAsType(TemPsy::OrderPattern),
      causes:Sequence(String) = orderPattern.block2.event.name,
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)) = self.loadDistances(orderPattern.block2.timeDistance),
      causeSize:Integer = causes->size(),
      effects:Sequence(String) = orderPattern.block1.event.name,
      effectDistances:Sequence(Tuple(which:Integer, value:Integer)) = self.loadDistances(orderPattern.block1.timeDistance),
      effectSize:Integer = effects->size()
  in
  if causeDistances->isEmpty() then
    if effectDistances->isEmpty() then
      if orderPattern.timeDistance->isEmpty() then
        if causeSize = 1 then
          let cause:String = causes->first() in
          if effectSize = 1 then
            let effect:String = effects->first() in
            self.checkPatternResponseOneOnePlain(subtrace, cause, effect)
          else
            self.checkPatternResponseOneManyPlain(subtrace, cause, effects)
          endif
        else
          if effectSize = 1 then
            let effect:String = effects->first() in
            self.checkPatternResponseManyOnePlain(subtrace, causes, effect)
          else
            self.checkPatternResponseManyManyPlain(subtrace, causes, effects)
          endif
        endif
      else
```

```
        if causeSize = 1 then
          let cause:String = causes->first() in
          if effectSize = 1 then
            let effect:String = effects->first() in
            self.checkPatternResponseOneOneMid(subtrace, cause, orderPattern.timeDistance, effect)
          else
            self.checkPatternResponseOneManyMid(subtrace, cause, orderPattern.timeDistance, effects)
          endif
        else
          if effectSize = 1 then
            let effect:String = effects->first() in
            self.checkPatternResponseManyOneMid(subtrace, causes, orderPattern.timeDistance, effect)
          else
            self.checkPatternResponseManyManyMid(subtrace, causes, orderPattern.timeDistance, effects)
          endif
        endif
      endif
    else
      if orderPattern.timeDistance->isEmpty() then
        if causeSize = 1 then
          let cause:String = causes->first() in
          self.checkPatternResponseOneManyRight(subtrace, cause, effects, effectDistances)
        else
          self.checkPatternResponseManyManyRight(subtrace, causes, effects, effectDistances)
        endif
      else
        if causeSize = 1 then
          let cause:String = causes->first() in
          self.checkPatternResponseOneManyMidRight(subtrace, cause, orderPattern.timeDistance, effects, effectDistances)
        else
          self.checkPatternResponseManyManyMidRight(subtrace, causes, orderPattern.timeDistance, effects, effectDistances)
        endif
      endif
    endif
  else
    if effectDistances->isEmpty() then
      if orderPattern.timeDistance->isEmpty() then
        if effectSize = 1 then
          let effect:String = effects->first() in
          self.checkPatternResponseManyOneLeft(subtrace, causes, causeDistances, effect)
        else
          self.checkPatternResponseManyManyLeft(subtrace, causes, causeDistances, effects)
        endif
      else
        if effectSize = 1 then
          let effect:String = effects->first() in
          self.checkPatternResponseManyOneLeftMid(subtrace, causes, causeDistances, orderPattern.timeDistance, effect)
        else
          self.checkPatternResponseManyManyLeftMid(subtrace, causes, causeDistances, orderPattern.timeDistance, effects)
        endif
      endif
    else
      if orderPattern.timeDistance->isEmpty() then
        self.checkPatternResponseManyManyLeftRight(subtrace, causes, causeDistances, effects, effectDistances)
      else
        self.checkPatternResponseManyManyLeftMidRight(subtrace, causes, causeDistances, orderPattern.timeDistance, effects, effectDistances)
      endif
    endif
  endif
endif

=======
def: checkPatternResponseOneOnePlain(subtrace:OrderedSet(trace::TraceElement), cause:String, effect:String):Boolean =
subtrace->iterate(
```

```
  elem:trace::TraceElement;
  result:Boolean = true
  |
  let e:String = elem.event in
  if e = cause then
    false
  else
    if e = effect then
      true
    else
      result
    endif
  endif
)

=======
def: checkPatternResponseOneOneAtLeastMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effect:String):Boolean =
subtrace->iterate(elem:trace::TraceElement;
  midCriticalInstant:Integer = 0
  |
  let e:String = elem.event in
  if e = cause then
    elem.timestamp + distance
  else
    if e = effect and elem.timestamp >= midCriticalInstant then
      0
    else
      midCriticalInstant
    endif
  endif
) = 0

=======
def: checkPatternResponseOneOneAtMostMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effect:String):Boolean =
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if iter.midCriticalInstant = 0 and e = cause then
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance}
    else
      if e = effect then
        if elem.timestamp <= iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0}
        else
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1} // violation
        endif
      else
        iter
      endif
    endif
  else
    iter
  endif
).midCriticalInstant = 0

=======
def: checkPatternResponseOneOneExactlyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effect:String):Boolean =
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer))
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}}
  |
  if iter.flag then
    let e:String = elem.event in
```

```
    if e = cause then
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.
           timestamp + distance)}
    else
      if e = effect and iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstants->first()
           then
        let t:Integer = elem.timestamp in
        if t = iter.midCriticalInstants->first() then
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding
               (t)}
        else
          Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants}
        endif
      else
        iter
      endif
    endif
  else
    iter
  endif
).midCriticalInstants->isEmpty()

def: checkPatternResponseOneOneMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:TemPsy::
     TimeDistance, effect:String):Boolean =
let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
if which = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseOneOneAtLeastMid(subtrace, cause, value, effect)
else
  if which = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseOneOneAtMostMid(subtrace, cause, value, effect)
  else
    self.checkPatternResponseOneOneExactlyMid(subtrace, cause, value, effect)
  endif
endif

=======
def: checkPatternResponseOneManyPlain(subtrace:OrderedSet(trace::TraceElement), cause:String, effects:Sequence(String
     )):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(
  elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, i2:Integer) = Tuple{flag:Boolean = true, i2:Integer = 1}
  |
  let e:String = elem.event in
  if e = cause then
    Tuple{flag:Boolean = false, i2:Integer = 1}
  else
    if not iter.flag then
      if e = effects->at(iter.i2) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = true, i2:Integer = 1}
        else
          Tuple{flag:Boolean = iter.flag, i2:Integer = iter.i2 + 1}
        endif
      else
        if e = firstEffect then
          Tuple{flag:Boolean = iter.flag, i2:Integer = 2}
        else
          Tuple{flag:Boolean = iter.flag, i2:Integer = 1}
        endif
      endif
    else
      iter
    endif
  endif
).flag
```

```

=======
def: checkPatternResponseOneManyAtLeastMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effects:Sequence(String)):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  let e:String = elem.event in
  if e = cause then // latest cause
    Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + distance, i2:Integer = 1}
  else
    if not iter.flag then
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        if iter.i2 = effectSize then // until effects->last(), the property is satisfied so far
          Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2 + 1}
        endif
      else
        if e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
        endif
      endif
    else
      iter
    endif
  endif
).flag

=======
def: checkPatternResponseOneManyAtMostMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effects:Sequence(String)):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  let e:String = elem.event in
  if iter.flag then
    if iter.midCriticalInstant = 0 then
      if e = cause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i2:Integer = 1}
      else
        iter
      endif
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i2:Integer = 1}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2 + 1}
        endif
      else
        if e = firstEffect then
          if elem.timestamp <= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
          else
```

```
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i2:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstant = 0

=======
def: checkPatternResponseOneManyExactlyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer, effects:Sequence(String)):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then
      let ct:Integer = elem.timestamp + distance in
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct), midCriticalInstant:Integer = ct, i2:Integer = 1}
    else
      if iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstant then
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          if iter.i2 = effectSize then
            if iter.midCriticalInstants->size() = 1 then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
            else
              let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i2:Integer = 1}
            endif
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2 + 1}
          endif
        else
          if e = firstEffect and elem.timestamp = iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
          else
            Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = null, i2:Integer = null}
          endif
        endif
      else
        iter
      endif
    endif
  else
    iter
  endif
).midCriticalInstants->isEmpty()

=======
def: checkPatternResponseOneManyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:TemPsy::TimeDistance, effects:Sequence(String)):Boolean =
```

```
let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
if which = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseOneManyAtLeastMid(subtrace, cause, value, effects)
else
  if which = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseOneManyAtMostMid(subtrace, cause, value, effects)
  else
    self.checkPatternResponseOneManyExactlyMid(subtrace, cause, value, effects)
  endif
endif

=======
def: checkPatternResponseManyOnePlain(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effect:
    String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, i1:Integer) = Tuple{flag:Boolean = true, i1:Integer = 1}
  |
  let e:String = elem.event in
  if iter.i1 > 1 and e = causes->at(iter.i1) then
    if iter.i1 = causeSize then
      Tuple{flag:Boolean = false, i1:Integer = 1}
    else
      Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1}
    endif
  else
    if e = firstCause then
      Tuple{flag:Boolean = iter.flag, i1:Integer = 2}
    else
      if e = effect then
        Tuple{flag:Boolean = true, i1:Integer = 1}
      else
        Tuple{flag:Boolean = iter.flag, i1:Integer = 1}
      endif
    endif
  endif
).flag

=======
def: checkPatternResponseManyOneAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    distance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(midCriticalInstant:Integer, i1:Integer) = Tuple{midCriticalInstant:Integer = 0, i1:Integer = 1}
  |
  let e:String = elem.event in
  if iter.i1 > 1 and e = causes->at(iter.i1) then
    if iter.i1 = causeSize then
      Tuple{midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1}
    else
      Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1}
    endif
  else
    if e = firstCause then
      Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
    else
      if e = effect and elem.timestamp >= iter.midCriticalInstant then
        Tuple{midCriticalInstant:Integer = 0, i1:Integer = 1}
      else
        Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
      endif
    endif
  endif
```

```
).midCriticalInstant = 0

=======
def: checkPatternResponseManyOneAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1} |
  let e:String = elem.event in
  if iter.flag then
    if iter.midCriticalInstant = 0 then
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1}
        endif
      else
        if e = firstCause then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
        endif
      endif
    else
      if e = effect then
        if elem.timestamp <= iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = 1}
        else
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null}
        endif
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
      endif
    endif
  else
    iter
  endif
).midCriticalInstant = 0

=======
def: checkPatternResponseManyOneExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:Integer, effect:String):Boolean =
let causeSize:Integer = causes->size(), firstCause:String = causes->first() in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1:Integer = 1}
  |
  if iter.flag then
    let e:String = elem.event, t:Integer = elem.timestamp in
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        let ct:Integer = t + distance in
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct), midCriticalInstant:Integer = ct, i1:Integer = 1}
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1}
      endif
    else
      if e = firstCause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
```

```
      else
        if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
          if t = iter.midCriticalInstant and e = effect then
            if iter.midCriticalInstants->size() = 1 then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
            else
              let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i1:Integer = 1}
            endif
          else
            Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = null, i1:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstants->isEmpty()

def: checkPatternResponseManyOneMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:TemPsy::TimeDistance, effect:String):Boolean =
let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
if which = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseManyOneAtLeastMid(subtrace, causes, value, effect)
else
  if which = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseManyOneAtMostMid(subtrace, causes, value, effect)
  else
    self.checkPatternResponseManyOneExactlyMid(subtrace, causes, value, effect)
  endif
endif

=======
def: checkPatternResponseManyManyPlain(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, i1:Integer, i2:Integer) = Tuple{flag:Boolean = true, i1:Integer = 1, i2:Integer = 1}
  |
  let e:String = elem.event in
  if iter.i2 = effectSize and e = lastEffect then
    Tuple{flag:Boolean = true, i1:Integer = 1, i2:Integer = 1}
  else
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, i1:Integer = 1, i2:Integer = 1}
      else
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = 2}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = 1}
```

```
        endif
      endif
    endif
  else
    if e = firstCause then
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = iter.i2 + 1}
      else
        if not iter.flag and e = firstEffect then
          Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = 2}
        else
          Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = 1}
        endif
      endif
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = iter.i2 + 1}
      else
        if not iter.flag and e = firstEffect then
          Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = 2}
        else
          Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = 1}
        endif
      endif
    endif
  endif
endif
).flag

=======
def: checkPatternResponseManyManyAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:Integer, effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1}
  |
  let e:String = elem.event in
  if iter.i2 = effectSize and e = lastEffect then
    Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1}
  else
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1, i2:Integer = 1}
      else
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1}
        else
          if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 2}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 1}
          endif
        endif
      endif
    else
      if e = firstCause then
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
```

```
          Integer = iter.i2 + 1}
        else
          if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 2}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 1}
          endif
        endif
      else
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = iter.i2 + 1}
        else
          if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1}
          endif
        endif
      endif
    endif
  endif
).flag

=======
def: checkPatternResponseManyManyAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:Integer, effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1}
  |
  let e:String = elem.event in
  if iter.flag then
    if iter.midCriticalInstant = 0 then
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1, i2:Integer = iter.i2}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2}
        endif
      else
        if e = firstCause then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = iter.i2}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = iter.i2}
        endif
      endif
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = iter.i1, i2:Integer = 1}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, i2:Integer = iter.i2 + 1}
        endif
```

```
      else
        if e = firstEffect then
          if elem.timestamp <= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, i2:Integer = 2}
          else
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, i2:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, i2:Integer = 1}
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstant = 0

=======
def: checkPatternResponseManyManyExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:Integer, effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1} |
  if iter.flag then
    let e:String = elem.event in
    if iter.i2 = effectSize and e = lastEffect then
      if iter.midCriticalInstants->size() = 1 then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1}
      else
        let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i1:Integer = 1, i2:Integer = 1}
      endif
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          let ct:Integer = elem.timestamp + distance in
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct), midCriticalInstant:Integer = ct, i1:Integer = 1, i2:Integer = 1}
        else
          if iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstant then
            if iter.i2 > 1 and e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1}
            else
              if e = firstEffect and elem.timestamp = iter.midCriticalInstant then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 2}
              else
                Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null}
              endif
            endif
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
```

```
                 midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 1}
          endif
        endif
      else
        if e = firstCause then
          if iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstant then
            if iter.i2 > 1 and e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                   midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = iter.i2 + 1}
            else
              if e = firstEffect and elem.timestamp = iter.midCriticalInstant then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                     midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 2}
              else
                Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                     midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null}
              endif
            endif
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                 midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 1}
          endif
        else
          if iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstant then
            if iter.i2 > 1 and e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                   midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = iter.i2 + 1}
            else
              if e = firstEffect and elem.timestamp = iter.midCriticalInstant then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                     midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2}
              else
                Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                     midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null}
              endif
            endif
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                 midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1}
          endif
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstants->isEmpty()

=======
def: checkPatternResponseManyManyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:
     TemPsy::TimeDistance, effects:Sequence(String)):Boolean =
let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
if which = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseManyManyAtLeastMid(subtrace, causes, value, effects)
else
  if which = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseManyManyAtMostMid(subtrace, causes, value, effects)
  else
    self.checkPatternResponseManyManyExactlyMid(subtrace, causes, value, effects)
  endif
endif

=======
def: checkPatternResponseOneManyRight(subtrace:OrderedSet(trace::TraceElement), cause:String, effects:Sequence(String
     ), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
```

```
in
subtrace->iterate(
  elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, i2:Integer, effectCriticalInstant:Integer) = Tuple{flag:Boolean = true, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if e = cause then
    Tuple{flag:Boolean = false, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
  else
    if not iter.flag then
      let t:Integer = elem.timestamp in
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = true, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        endif
      else
        if e = firstEffect then
          Tuple{flag:Boolean = iter.flag, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
        else
          Tuple{flag:Boolean = iter.flag, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    else
      iter
    endif
  endif
).flag

=======
def: checkPatternResponseManyManyRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, i1:Integer, i2:Integer, effectCriticalInstant:Integer) = Tuple{flag:Boolean = true, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if iter.i2 = effectSize and e = lastEffect and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->last().which) then
    Tuple{flag:Boolean = true, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
  else
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
      else
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer =
```

```
                    t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    else
      if e = firstCause then
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      else
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    endif
  endif
).flag

=======
def: checkPatternResponseOneManyAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if e = cause then // latest cause
    Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + midDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
  else
    if not iter.flag then
      let t:Integer = elem.timestamp in
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
```

```
        else
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i,
               effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        endif
      else
        if e = firstEffect and t >= iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2,
               effectCriticalInstant:Integer = t + secondEffectDistance}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1,
               effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    else
      iter
    endif
  endif
).flag

=======
def: checkPatternResponseOneManyAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:
     Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if iter.flag then
    if iter.midCriticalInstant = 0 then
      if e = cause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i2:Integer = 1,
             effectCriticalInstant:Integer = iter.effectCriticalInstant}
      else
        iter
      endif
    else
      let t:Integer = elem.timestamp in
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at
           (iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:
               Integer = iter.effectCriticalInstant}
        else
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i,
               effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        endif
      else
        if e = firstEffect then
          if t <= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2,
                 effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i2:Integer = null, effectCriticalInstant:
                 Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1,
               effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    endif
  else
```

```
    iter
  endif
).midCriticalInstant = 0

=======
def: checkPatternResponseOneManyExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event in
    if e = cause then
      let ct:Integer = elem.timestamp + midDistance in
      Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct), midCriticalInstant:Integer = ct, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
    else
      let t:Integer = elem.timestamp in
      if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
          if iter.i2 = effectSize then
            if iter.midCriticalInstants->size() = 1 then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            else
              let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
            endif
          else
            let i:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
          endif
        else
          if e = firstEffect and t = iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
          endif
        endif
      else
        iter
      endif
    endif
  else
    iter
  endif
).midCriticalInstants->isEmpty()

=======
def: checkPatternResponseOneManyMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:TemPsy::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
```

```
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator = midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseOneManyAtLeastMidRight(subtrace, cause, midValue, effects, effectDistances)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseOneManyAtMostMidRight(subtrace, cause, midValue, effects, effectDistances)
  else
    self.checkPatternResponseOneManyExactlyMidRight(subtrace, cause, midValue, effects, effectDistances)
  endif
endif

=======
def: checkPatternResponseManyManyAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):
    Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:
      Integer = 0}
  |
  let e:String = elem.event in
  if iter.i2 = effectSize and e = lastEffect and self.compare(elem.timestamp, iter.effectCriticalInstant,
      effectDistances->last().which) then
    Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1,
        effectCriticalInstant:Integer = iter.effectCriticalInstant}
  else
    if iter.i1 > 1 and e = causes->at(iter.i1) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, i2:
            Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
      else
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
            effectDistances->at(iter.i2).which) then
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1
              + 1, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        else
          if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
                i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
                i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    else
      if e = firstCause then
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
            effectDistances->at(iter.i2).which) then
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
              Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        else
          if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
                Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
```

```
                Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      else
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
          let i:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
        else
          if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    endif
  endif
).flag

=======
def: checkPatternResponseManyManyAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if iter.flag then
    if iter.midCriticalInstant = 0 then
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      else
        if e = firstCause then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    else
      let t:Integer = elem.timestamp in
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = iter.i1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          let i:Integer = iter.i2 + 1 in
```

```
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
      endif
    else
      if e = firstEffect then
        if t <= iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
        else
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
        endif
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
      endif
    endif
  endif
else
  iter
endif
).midCriticalInstant = 0

=======
def: checkPatternResponseManyManyExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event, t:Integer = elem.timestamp in
    if iter.i2 = effectSize and e = lastEffect then
      if iter.midCriticalInstants->size() = 1 then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
      else
        let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
      endif
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) then
        if iter.i1 = causeSize then
          let ct:Integer = t + midDistance in
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct), midCriticalInstant:Integer = ct, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
            if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
              let i:Integer = iter.i2 + 1 in
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
```

```
          else
            if e = firstEffect and t = iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                   midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 2,
                   effectCriticalInstant:Integer = t + secondEffectDistance}
            else
              Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                   midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null, effectCriticalInstant:
                   Integer = null}
            endif
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
               midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 1,
               effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    else
      if e = firstCause then
        if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
               effectDistances->at(iter.i2).which) then
            let i:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                 midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = i,
                 effectCriticalInstant:Integer = t + effectDistances->at(i).value}
          else
            if e = firstEffect and t = iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                   midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 2,
                   effectCriticalInstant:Integer = t + secondEffectDistance}
            else
              Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                   midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null, effectCriticalInstant:
                   Integer = null}
            endif
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
               midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 1,
               effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      else
        if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
          if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
               effectDistances->at(iter.i2).which) then
            let i:Integer = iter.i2 + 1 in
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                 midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = i,
                 effectCriticalInstant:Integer = t + effectDistances->at(i).value}
          else
            if e = firstEffect and t = iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                   midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2,
                   effectCriticalInstant:Integer = t + secondEffectDistance}
            else
              Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                   midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null, effectCriticalInstant:
                   Integer = null}
            endif
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
               midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1,
               effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    endif
  endif
```

```
  else
    iter
  endif
).midCriticalInstants->isEmpty()

=======
def: checkPatternResponseManyManyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), midDistance:TemPsy::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator = midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseManyManyAtLeastMidRight(subtrace, causes, midValue, effects, effectDistances)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseManyManyAtMostMidRight(subtrace, causes, midValue, effects, effectDistances)
  else
    self.checkPatternResponseManyManyExactlyMidRight(subtrace, causes, midValue, effects, effectDistances)
  endif
endif

=======
def: checkPatternResponseManyOneLeft(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, i1:Integer, causeCriticalInstant:Integer) = Tuple{flag:Boolean = true, i1:Integer = 1, causeCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
    if iter.i1 = causeSize then
      Tuple{flag:Boolean = false, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
    else
      let i:Integer = iter.i1 + 1 in
      Tuple{flag:Boolean = iter.flag, i1:Integer = i, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value}
    endif
  else
    if e = firstCause then
      Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance}
    else
      if e = effect then
        Tuple{flag:Boolean = true, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
      else
        Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
      endif
    endif
  endif
).flag

=======
def: checkPatternResponseManyManyLeft(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effects:Sequence(String)):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, i1:Integer, causeCriticalInstant:Integer, i2:Integer) = Tuple{flag:Boolean = true, i1:
```

```
Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1}
  |
  let e:String = elem.event in
  if iter.i2 = effectSize and e = lastEffect then
    Tuple{flag:Boolean = true, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
  else
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
      else
        let i:Integer = iter.i1 + 1 in
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, i1:Integer = i, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = iter.i2 + 1}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = i, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = 2}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = i, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = 1}
          endif
        endif
      endif
    else
      if e = firstCause then
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
          endif
        endif
      else
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
          endif
        endif
      endif
    endif
  endif
).flag

=======
def: checkPatternResponseManyOneLeftAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer) = Tuple{midCriticalInstant:Integer
```

```
    = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
  |
  let e:String = elem.event, t:Integer = elem.timestamp in
  if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
    if iter.i1 = causeSize then
      Tuple{midCriticalInstant:Integer = t + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
    else
      let i:Integer = iter.i1 + 1 in
      Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i, causeCriticalInstant:Integer = t + causeDistances->at(i).value}
    endif
  else
    if e = firstCause then
      Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = t + secondCauseDistance}
    else
      if e = effect and t >= iter.midCriticalInstant then
        Tuple{midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
      else
        Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
      endif
    endif
  endif
).midCriticalInstant = 0

=======
def: checkPatternResponseManyOneLeftAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if iter.flag then
    if iter.midCriticalInstant = 0 then
      let t:Integer = elem.timestamp in
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = t + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
        else
          let i:Integer = iter.i1 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i, causeCriticalInstant:Integer = t + causeDistances->at(i).value}
        endif
      else
        if e = firstCause then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = t + secondCauseDistance}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
        endif
      endif
    else
      if e = effect then
        if elem.timestamp <= iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
```

```
        else
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null}
        endif
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
      endif
    endif
  else
    iter
  endif
).midCriticalInstant = 0

=======
def: checkPatternResponseManyOneLeftExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event, t:Integer = elem.timestamp in
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
      if iter.i1 = causeSize then
        let ct:Integer = t + midDistance in
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct), midCriticalInstant:Integer = ct, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
      else
        let i:Integer = iter.i1 + 1 in
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i, causeCriticalInstant:Integer = t + causeDistances->at(i).value}
      endif
    else
      if e = firstCause then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = t + secondCauseDistance}
      else
        if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
          if t = iter.midCriticalInstant and e = effect then
            if iter.midCriticalInstants->size() = 1 then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
            else
              let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
            endif
          else
            Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
```

```
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstants->isEmpty()

=======
def: checkPatternResponseManyOneLeftMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
     causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TemPsy::TimeDistance, effect:String):
     Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator = midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseManyOneLeftAtLeastMid(subtrace, causes, causeDistances, midValue, effect)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseManyOneLeftAtMostMid(subtrace, causes, causeDistances, midValue, effect)
  else
    self.checkPatternResponseManyOneLeftExactlyMid(subtrace, causes, causeDistances, midValue, effect)
  endif
endif

=======
def: checkPatternResponseManyManyLeftAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
     causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)):
     Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
      Integer = 1}
  |
  let e:String = elem.event in
  if iter.i2 = effectSize and e = lastEffect then
    Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
        causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
  else
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
        causeDistances->at(iter.i1).which) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
            causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
      else
        let i:Integer = iter.i1 + 1 in
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i,
              causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = iter.i2 + 1}
        else
          if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i,
                causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = 2}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i,
                causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = 1}
          endif
        endif
      endif
    else
      if e = firstCause then
        if iter.i2 > 1 and e = effects->at(iter.i2) then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
```

```
        causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
    else
      if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
            causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
            causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
      endif
    endif
  else
    if iter.i2 > 1 and e = effects->at(iter.i2) then
      Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
          causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
    else
      if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
            causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
            causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
      endif
    endif
  endif
 endif
endif
).flag

=======
def: checkPatternResponseManyManyLeftAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)):
    Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
      Integer = 1} |
  let e:String = elem.event in
  if iter.flag then
    if iter.midCriticalInstant = 0 then
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
          causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
              causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2}
        else
          let i:Integer = iter.i1 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i,
              causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = iter.i2}
        endif
      else
        if e = firstCause then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
              causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
              causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2}
        endif
      endif
    else
      if iter.i2 > 1 and e = effects->at(iter.i2) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = iter.i1, causeCriticalInstant:
```

```
        Integer = iter.causeCriticalInstant, i2:Integer = 1}
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
      endif
    else
      if e = firstEffect then
        if elem.timestamp <= iter.midCriticalInstant then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
        else
          Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null}
        endif
      else
        Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
      endif
    endif
  endif
else
  iter
endif
).midCriticalInstant = 0

=======
def: checkPatternResponseManyManyLeftExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)): Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last()
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1} |
  if iter.flag then
    let e:String = elem.event, t:Integer = elem.timestamp in
    if iter.i2 = effectSize and e = lastEffect then
      if iter.midCriticalInstants->size() = 1 then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
      else
        let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
      endif
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          let ct:Integer = t + midDistance in
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct), midCriticalInstant:Integer = ct, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
        else
          let i:Integer = iter.i1 + 1 in
          if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
            if iter.i2 > 1 and e = effects->at(iter.i2) then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i, causeCriticalInstant:Integer
```

```
                  = elem.timestamp + causeDistances->at(i).value, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and t = iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                  midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i, causeCriticalInstant:
                  Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = 2}
            else
              Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                  midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                  Integer = null}
            endif
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
              midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i, causeCriticalInstant:Integer =
               elem.timestamp + causeDistances->at(i).value, i2:Integer = 1}
        endif
      endif
    else
      if e = firstCause then
        if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer
                 = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and t = iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                  midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:
                  Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
            else
              Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                  midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                  Integer = null}
            endif
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
              midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer =
               elem.timestamp + secondCauseDistance, i2:Integer = 1}
        endif
      else
        if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
          if iter.i2 > 1 and e = effects->at(iter.i2) then
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer
                 = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
          else
            if e = firstEffect and t = iter.midCriticalInstant then
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                  midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:
                  Integer = iter.causeCriticalInstant, i2:Integer = 2}
            else
              Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                  midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                  Integer = null}
            endif
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
              midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer =
               iter.causeCriticalInstant, i2:Integer = 1}
        endif
      endif
    endif
  endif
else
  iter
endif
```

```
).midCriticalInstants->isEmpty()

=======
def: checkPatternResponseManyManyLeftMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TemPsy::TimeDistance, effects:Sequence(String)):Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator = midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseManyManyLeftAtLeastMid(subtrace, causes, causeDistances, midValue, effects)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseManyManyLeftAtMostMid(subtrace, causes, causeDistances, midValue, effects)
  else
    self.checkPatternResponseManyManyLeftExactlyMid(subtrace, causes, causeDistances, midValue, effects)
  endif
endif

=======
def: checkPatternResponseManyManyLeftRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, i1:Integer, causeCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer) = Tuple{flag:Boolean = true, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if iter.i2 = effectSize and e = lastEffect and self.compare(elem.timestamp, iter.effectCriticalInstant, effectDistances->last().which) then
    Tuple{flag:Boolean = true, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
  else
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
      else
        let t:Integer = elem.timestamp, i11:Integer = iter.i1 + 1 in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, i1:Integer = i11, causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = i22, effectCriticalInstant:Integer = t + causeDistances->at(i22).value}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = i11, causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    else
      if e = firstCause then
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
```

```
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = t + secondCauseDistance, i2:
              Integer = i22, effectCriticalInstant:Integer = t + causeDistances->at(i22).value}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = t + secondCauseDistance,
                i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp +
                secondCauseDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      else
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
            at(iter.i2).which) then
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
              i2:Integer = i22, effectCriticalInstant:Integer = t + causeDistances->at(i22).value}
        else
          if not iter.flag and e = firstEffect then
            Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
                 i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
                 i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    endif
  endif
).flag


=======
def: checkPatternResponseManyManyLeftAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String
    ), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String),
    effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer,
      effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
      Integer = 1, effectCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if iter.i2 = effectSize and e = lastEffect and self.compare(elem.timestamp, iter.effectCriticalInstant,
      effectDistances->last().which) then
    Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
        causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer =
        iter.effectCriticalInstant}
  else
    if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
        causeDistances->at(iter.i1).which) then
      if iter.i1 = causeSize then
        Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
            causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer
            = iter.effectCriticalInstant}
      else
        let t:Integer = elem.timestamp, i11:Integer = iter.i1 + 1 in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
```

```
            at(iter.i2).which) then
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
              causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = i22,
              effectCriticalInstant:Integer = t + causeDistances->at(i22).value}
        else
          if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
                causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = 2,
                effectCriticalInstant:Integer = t + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
                causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = 1,
                effectCriticalInstant:Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    else
      if e = firstCause then
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
            at(iter.i2).which) then
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
              causeCriticalInstant:Integer = t + secondCauseDistance, i2:Integer = i22, effectCriticalInstant:
              Integer = elem.timestamp + causeDistances->at(i22).value}
        else
          if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                causeCriticalInstant:Integer = t + secondCauseDistance, i2:Integer = 2, effectCriticalInstant:
                Integer = elem.timestamp + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                causeCriticalInstant:Integer = t + secondCauseDistance, i2:Integer = 1, effectCriticalInstant:
                Integer = iter.effectCriticalInstant}
          endif
        endif
      else
        let t:Integer = elem.timestamp in
        if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
            at(iter.i2).which) then
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
              causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:
              Integer = elem.timestamp + causeDistances->at(i22).value}
        else
          if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:
                Integer = elem.timestamp + secondEffectDistance}
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:
                Integer = iter.effectCriticalInstant}
          endif
        endif
      endif
    endif
  endif
).flag

=======
def: checkPatternResponseManyManyLeftAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String)
    , causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String),
    effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
```

```
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  let e:String = elem.event in
  if iter.flag then
    if iter.midCriticalInstant = 0 then
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          let i11:Integer = iter.i1 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      else
        if e = firstCause then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    else
      let t:Integer = elem.timestamp in
      if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
        if iter.i2 = effectSize then
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = iter.i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          let i22:Integer = iter.i2 + 1 in
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = t + effectDistances->at(i22).value}
        endif
      else
        if e = firstEffect then
          if t <= iter.midCriticalInstant then
            Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = t + effectDistances->at(2).value}
          else
            Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
          endif
        else
          Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        endif
      endif
    endif
  else
    iter
```

```
  endif
).midCriticalInstant = 0

=======
def: checkPatternResponseManyManyLeftExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let
  causeSize:Integer = causes->size(),
  firstCause:String = causes->first(),
  secondCauseDistance:Integer = causeDistances->at(2).value,
  effectSize:Integer = effects->size(),
  firstEffect:String = effects->first(),
  lastEffect:String = effects->last(),
  secondEffectDistance:Integer = effectDistances->at(2).value
in
subtrace->iterate(elem:trace::TraceElement;
  iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
  = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
  |
  if iter.flag then
    let e:String = elem.event, t:Integer = elem.timestamp in
    if iter.i2 = effectSize and e = lastEffect then
      if iter.midCriticalInstants->size() = 1 then
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
      else
        let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
        Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
      endif
    else
      if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(iter.i1).which) then
        if iter.i1 = causeSize then
          let ct:Integer = t + midDistance in
          Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct), midCriticalInstant:Integer = ct, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
        else
          let i11:Integer = iter.i1 + 1 in
          if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
            if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at(iter.i2).which) then
              let i22:Integer = iter.i2 + 1 in
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = i22, effectCriticalInstant:Integer = t + effectDistances->at(i22).value}
            else
              if e = firstEffect and t = iter.midCriticalInstant then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
              else
                Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
              endif
            endif
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
```

```
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11, causeCriticalInstant:Integer
                 = t + causeDistances->at(i11).value, i2:Integer = 1, effectCriticalInstant:Integer = iter.
                effectCriticalInstant}
          endif
        endif
      else
        if e = firstCause then
          if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
            if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
                effectDistances->at(iter.i2).which) then
              let i22:Integer = iter.i2 + 1 in
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                  midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer
                   = elem.timestamp + secondCauseDistance, i2:Integer = i22, effectCriticalInstant:Integer = t +
                  effectDistances->at(i22).value}
            else
              if e = firstEffect and t = iter.midCriticalInstant then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                    midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:
                    Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2, effectCriticalInstant:Integer =
                    t + secondEffectDistance}
              else
                Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                    midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                    Integer = null, effectCriticalInstant:Integer = null}
              endif
            endif
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer =
                 elem.timestamp + secondCauseDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.
                effectCriticalInstant}
          endif
        else
          if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
            if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
                effectDistances->at(iter.i2).which) then
              let i22:Integer = iter.i2 + 1 in
              Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                  midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer
                   = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:Integer = t +
                  effectDistances->at(i22).value}
            else
              if e = firstEffect and t = iter.midCriticalInstant then
                Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                    midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:
                    Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = t +
                    secondEffectDistance}
              else
                Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                    midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                    Integer = null, effectCriticalInstant:Integer = null}
              endif
            endif
          else
            Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer =
                 iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.
                effectCriticalInstant}
          endif
        endif
      endif
    endif
  else
    iter
  endif
).midCriticalInstants->isEmpty()

=======
```

```
def: checkPatternResponseManyManyLeftMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TemPsy::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator = midDistance.comparingOperator in
if midWhich = TemPsy::ComparingOperator::ATLEAST then
  self.checkPatternResponseManyManyLeftAtLeastMidRight(subtrace, causes, causeDistances, midValue, effects, effectDistances)
else
  if midWhich = TemPsy::ComparingOperator::ATMOST then
    self.checkPatternResponseManyManyLeftAtMostMidRight(subtrace, causes, causeDistances, midValue, effects, effectDistances)
  else
    self.checkPatternResponseManyManyLeftExactlyMidRight(subtrace, causes, causeDistances, midValue, effects, effectDistances)
  endif
endif
```