

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

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 technology¹), 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-

¹www.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 *TempPsy* (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 *TempPsy* 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 TEMPSEY-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.

*TempPsy*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 *TempPsy*, 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, *TempPsy* 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. *TempPsy* 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 *TempPsy* into the SoftwareAG ARIS modeling tool [60], and its analysts have started using it to annotate business process models with *TempPsy* specifications. In this paper, we show the application of *TempPsy* for the specification of an excerpt of a business process extracted from the case study developed with our partner.

²The 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 *TempPsy* properties into OCL constraints defined over this trace model. This mapping is optimized based on the structure of the *TempPsy* property to check, in order to achieve better performance. More specifically, we show how the problem of checking a *TempPsy* property over an execution trace (i.e., the *TempPsy* trace checking problem) can be reduced to evaluating an OCL constraint (derived from the *TempPsy* 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 TEMPSPY-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 TEMPSPY-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 TEMPSPY-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 TEMPSPY-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 *TempPsy*, presenting its syntax and its (informal) semantics. In section 4 we show the application of *TempPsy* in a case study in the domain of eGovernment. Section 5 presents the formal semantics of *TempPsy*. Section 6 describes our model-driven approach for trace checking of *TempPsy* properties. Section 7 reports on the evaluation conducted with TEMPSPY-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 *TempPsy* language.

This catalogue³ 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

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

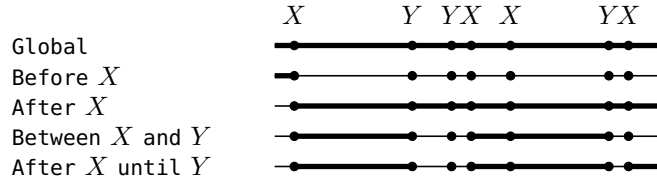


Fig. 1: Scopes in the catalogue of property specification patterns in [27]

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 *TempPsy*.

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

3 The *TempPsy* 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 *TempPsy* 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 *TempPsy* 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 *TempPsy* 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 *TempPsy* 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. *TempPsy* 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⟩ ‘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 $\langle TemPsyBlock \rangle$ comprises a set of conjuncted $\langle TemPsyExpression \rangle$ s. Each *TemPsy* expression starts with an optional ‘temporal’ keyword plus an alphanumeric identifier, followed by a $\langle Scope \rangle$ and a $\langle Pattern \rangle$. The keywords indicating the five $\langle Scope \rangle$ s identify univocally the corresponding scopes from [27] (see section 2). As for the $\langle Pattern \rangle$ 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 $\langle Scope \rangle$ s and $\langle Pattern \rangle$ s refer to the concept of $\langle Event \rangle$. We assume that an $\langle Event \rangle$ is rep-

resented 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. $\langle Scope \rangle$ s contain boundaries (expressed with $\langle Boundary1 \rangle$ or $\langle Boundary2 \rangle$) that denote a specific occurrence of an event as a boundary, possibly with a time distance; notice that $\langle Boundary2 \rangle$ represents a syntactic restriction of $\langle Boundary1 \rangle$. Chains of events, used in precedence and response patterns, are defined as $\langle EventChainExp \rangle$, which denotes a comma-separated list of events, possibly with a time distance ($\langle TimeDistanceExp \rangle$) 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 property⁴ 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*

⁴These 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.

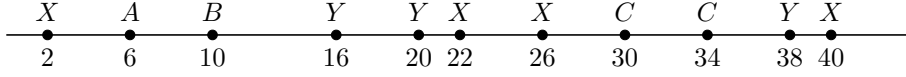


Fig. 3: An event trace on which to evaluate the properties described in section 3.4; events are above the line, timestamps below

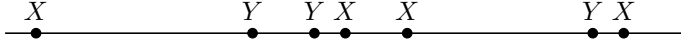


Fig. 4: A sample trace for the description of scopes

- 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 *TempPsy* 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 *TempPsy* 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 *TempPsy* 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 *TempPsy* 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 general⁵, 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.

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

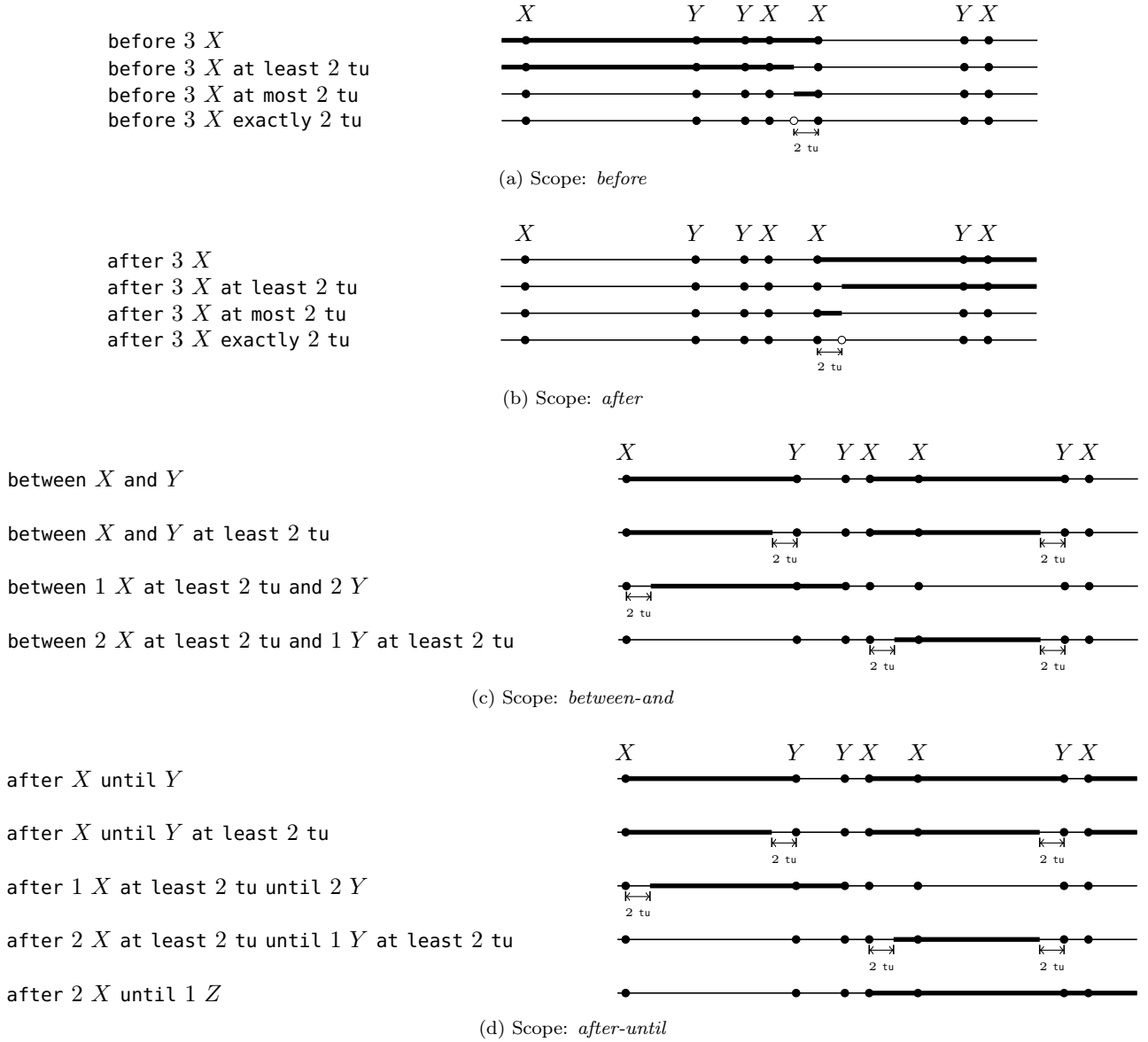


Fig. 5: Examples of *TemPsy* scopes

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-

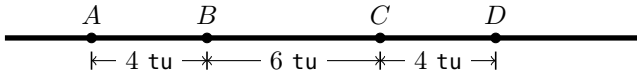


Fig. 6: Example trace for illustrating the precedence and response patterns

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-

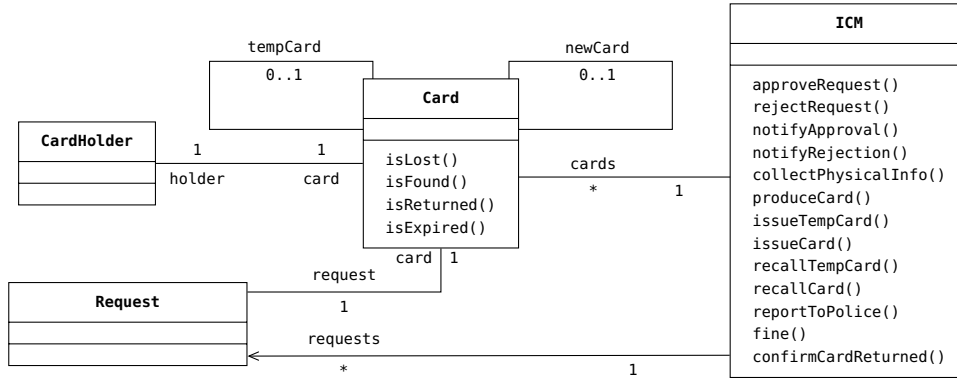


Fig. 7: Conceptual model of the ICM business process

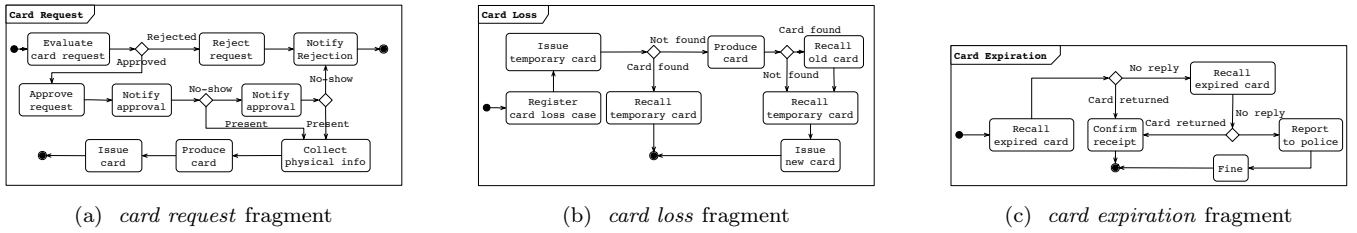


Fig. 8: Activity diagrams of three fragments of the ICM business process

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 *TempPsy* 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 *TempPsy* as follows:

```

1 temporal L1:
2 after Card.isLost
3 at most 24*3600 tu
4 ICM.recallTempCard
5 responding at most 10*24*3600 tu
6 ICM.issueTempCard
7 temporal L2:
8 after Card.isLost
9 ICM.produceCard
10 preceding at least 2*24*3600 tu
11 ICM.issueCard
12 temporal L3:
13 after Card.isLost
14 until ICM.issueCard

```

```

15 ICM.recallCard,
16 ICM.recallTempCard,
17 responding at most 3*24*3600 tu
18 ICM.produceCard,
19 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:

```

1 temporal E1:
2   after Card.isExpired
3   until Card.isReturned
4   eventually at most 2 ICM.recallCard
5 temporal E2:
6   after 2 ICM.recallCard
7   at least 5*24*3600 tu
8   until Card.isReturned
9   ICM.fine
10  responding
11  ICM.reportToPolice
12 temporal E3:
13  globally
14  ICM.confirmCardReturned
15  responding at most 24*3600 tu
16  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 pro-

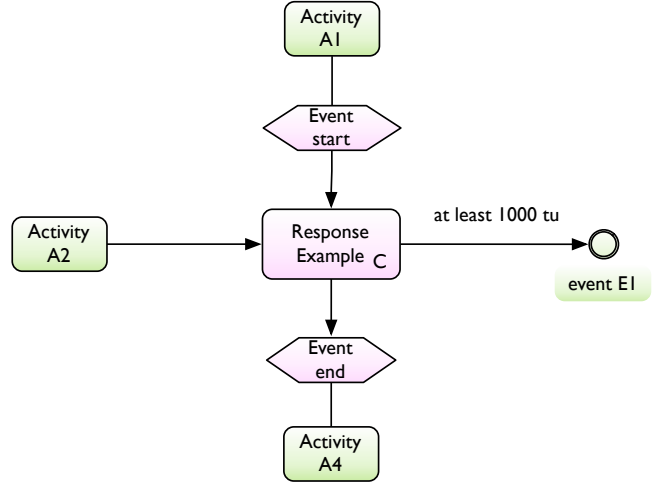


Fig. 9: Example of the graphical notation for *TempSy*

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

cess 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 framework⁶; 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.

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

5 Formal Semantics of *TempPsy*

This section presents the formal semantics of *TempPsy*, 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 $e_1, t_1, e_2, \dots, t_{m-1}, e_m$. 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 $t_i = \# \bowtie_i \delta_i \text{ tu}$ with $\delta_i \in \mathbb{N}^+$ and $\bowtie_i \in \{\text{at least, at most, exactly}\}$; when t_i is undefined we use the notation $t_i = \perp$. Function $\text{len}(EC)$ 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, \dots, 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 $\tau : \mathbb{N} \rightarrow \mathbb{N}$, 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 $\tau(0), \tau(1), \dots, \tau(n-1)$ 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 $\{0, \dots, n-1\}$ if defined or be equal to $\{\perp\}$ if undefined; $\alpha, \alpha', \beta', \gamma, \theta, \theta', \eta, \eta'$ be auxiliary variables ranging over $\{0, \dots, n-1\}$.

Scopes. Let \mathbb{S} be the set of scopes that can be derived from the non-terminal $\langle \text{Scope} \rangle$ 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 $\phi_{[s]}(\lambda) : \Lambda \rightarrow 2^\Lambda$ as follows:
globally: $\phi_{[\text{globally}]}(\lambda) = \{\lambda\}$

before:

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

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

after:

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

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

between-and:

- $\phi_{[\text{between } e_1 \text{ and } e_2]}(\lambda) = \{\lambda(\alpha_k + 1 : \beta_k - 1) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1\}$
- $\phi_{[\text{between } e_1 \text{ and } e_2 \text{ at least } d \text{ tu}]}(\lambda) = \{\lambda(\alpha_k + 1 : \beta'_k) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \beta'_k = \max(\{\gamma \mid td(\gamma, \beta_k) \geq d\})\}$
- $\phi_{[\text{between } e_1 \text{ at least } b \text{ tu and } e_2]}(\lambda) = \{\lambda(\alpha'_k : \beta_k - 1) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \alpha'_k = \min(\{\gamma \mid td(\alpha_k, \gamma) \geq b\})\}$
- $\phi_{[\text{between } e_1 \text{ at least } b \text{ tu and } e_2 \text{ at least } d \text{ tu}]}(\lambda) = \{\lambda(\alpha'_k : \beta'_k) \mid \forall k \geq 0, \alpha_k < \beta_k < \alpha_{k+1}, \lambda(\alpha_k) = e_1, \lambda(\beta_k) = e_2, \forall j, \alpha_k < j < \beta_k, \lambda(j) \neq e_2, \forall i, \beta_k < i < \alpha_{k+1}, \lambda(i) \neq e_1, \alpha'_k = \min(\{\gamma \mid td(\alpha_k, \gamma) \geq b\}), \beta'_k = \max(\{\gamma \mid td(\gamma, \beta_k) \geq d\})\}$
- $\phi_{[\text{between } a \text{ e}_1 \text{ and } c \text{ e}_2]}(\lambda) = \{\lambda(\alpha + 1 : \beta - 1) \mid \lambda(\alpha) = e_1, \#(\lambda, 0, \alpha, e_1) = x, \lambda(\beta) = e_2, \#(\lambda, \alpha + 1, \beta, e_2) = y\}$
- $\phi_{[\text{between } a \text{ e}_1 \text{ and } c \text{ e}_2 \text{ at least } d \text{ tu}]}(\lambda) = \{\lambda(\alpha + 1 : \beta') \mid \lambda(\alpha) = e_1, \#(\lambda, 0, \alpha, e_1) = x, \lambda(\beta) = e_2, \#(\lambda, \alpha + 1, \beta, e_2) = y, \beta' = \max(\{\gamma \mid td(\gamma, \beta) \geq d\})\}$
- $\phi_{[\text{between } a \text{ e}_1 \text{ at least } b \text{ tu and } c \text{ e}_2]}(\lambda) = \{\lambda(\alpha' : \beta - 1) \mid \lambda(\alpha) = e_1, \#(\lambda, 0, \alpha, e_1) = x, \lambda(\beta) = e_2, \#(\lambda, \alpha + 1, \beta, e_2) = y, \alpha' = \min(\{\gamma \mid td(\alpha, \gamma) \geq b\})\}$

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

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

after-until:

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

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

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 $EC = e_1, t_1, e_2, \dots, t_{m-1}, e_m$, we have $match(\lambda, EC, i) = true$, with $i, 0 \leq i \leq n - m$, if there exist $i_1, i_2, \dots, i_m \in \{0, \dots, n - 1\}$, such that $i_1 = i, i_{k+1} = i_k + 1, 1 \leq k \leq m - 1, \lambda(i_1) = e_1, \lambda(i_2) = e_2, \dots, \lambda(i_m) = e_m$ and for all $j, 1 \leq j \leq m - 1$, such that $t_j \neq \perp$, we have:

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

For an events chain $EC = e_1, t_1, e_2, \dots, t_{m-1}, e_m$ we also define two auxiliary functions $first(\lambda, EC, i)$ 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 $\psi_{[p]}(\lambda) : \Lambda \rightarrow \{true, false\}$ defined as follows:

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

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

existence:

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

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

precedence:

- $\psi_{[EC_1 \text{ preceding } EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n - 1, match(\lambda, EC_2, i) \Rightarrow \exists j, 0 \leq j \leq i - len(EC_1), match(\lambda, EC_1, j)$
- $\psi_{[EC_1 \text{ preceding } \bowtie \ b \text{ tu } EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n - 1, match(\lambda, EC_2, i) \Rightarrow \exists j, 0 \leq j \leq i - len(EC_1), match(\lambda, EC_1, j) \text{ and } (first(\lambda, EC_2, i) - last(\lambda, EC_1, j)) \Delta \ b$
 $\text{where } \Delta = \begin{cases} \geq, & \text{if } \bowtie = \text{at least;} \\ \leq, & \text{if } \bowtie = \text{at most;} \\ =, & \text{if } \bowtie = \text{exactly.} \end{cases}$

response:

- $\psi_{[EC_1 \text{ responding } EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n - 1, match(\lambda, EC_2, i) \Rightarrow \exists j, i + len(EC_2) \leq j \leq n - 1, match(\lambda, EC_1, j)$
- $\psi_{[EC_1 \text{ responding } \bowtie \ b \text{ tu } EC_2]}(\lambda) \Leftrightarrow \forall i, 0 \leq i < n - 1, match(\lambda, EC_2, i) \Rightarrow \exists j, i + len(EC_2) \leq j \leq n - 1, match(\lambda, EC_1, j) \text{ and } (first(\lambda, EC_1, i) - last(\lambda, EC_2, j)) \Delta \ b$
 $\text{where } \Delta = \begin{cases} \geq, & \text{if } \bowtie = \text{at least;} \\ \leq, & \text{if } \bowtie = \text{at most;} \\ =, & \text{if } \bowtie = \text{exactly.} \end{cases}$

Temporal Expression. The semantics over a trace λ of a temporal expression derived from the non-terminal $\langle TempPsyExpression \rangle$ 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 *TempPsy* properties

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

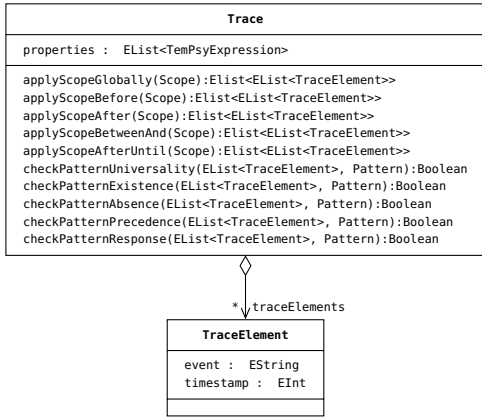


Fig. 10: Conceptual model for execution traces

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 *TempPsy* 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 *TempPsy* 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 *TempPsyExpressions*⁷, representing the properties to be checked on the trace.

We have defined some side-effect-free operations in OCL

⁷Class *TempPsyExpression* belongs to the meta-model of the language (not shown here for space reasons) and represents objects corresponding to the non-terminal $\langle TempPsyExpression \rangle$ of the grammar shown in Fig. 2.

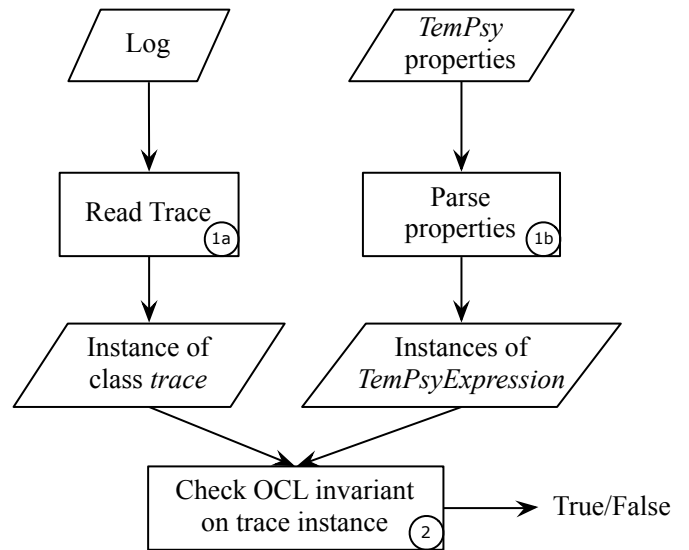


Fig. 11: Overview of the approach

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 *TempPsy* 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 *TempPsy* properties are parsed and converted (step 1b) to instances of class *TempPsyExpression*.

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 *TempPsy* properties provided in input.

We have defined this invariant on the *Trace* class, as shown in Fig. 12. For every *TempPsy* 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.

```

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

```

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 elements⁸ 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.

⁸In 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*

- 1 $X \leftarrow$ event name of the scope boundary
- 2 $m \leftarrow$ index of the specific occurrence of event X
- 3 $op \leftarrow$ comparison operator of the constraint on time distance
- 4 $n \leftarrow$ time distance from the m -th occurrence of X
- 5 $result \leftarrow []$, $segment \leftarrow []$
- 6 **if** $m = \text{null}$ **then** $m \leftarrow 1$
- 7 $t \leftarrow$ timestamp of the m -th occurrence of event X
- 8 **if** $t \neq \text{null}$ **then**
- 9 **switch** op **do**
- 10 **case** “at least” **do**
- 11 $segment \leftarrow$ trace elements with timestamp t' satisfying $t' \leq t - n$
- 12 **end**
- 13 **case** “at most” **do**
- 14 $segment \leftarrow$ trace elements with timestamp t' satisfying $t - n \leq t' < t$
- 15 **end**
- 16 **case** “exactly” **do**
- 17 $segment \leftarrow$ trace elements with timestamp equal to $t - n$
- 18 **end**
- 19 **otherwise do**
- 20 $segment \leftarrow$ trace elements with timestamp t' satisfying $t' < t$
- 21 **end**
- 22 **end**
- 23 **end**
- 24 $result.append(segment)$
- 25 **return** $result$

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 assign-

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 $X \leftarrow$ event name of the left boundary
- 2 $Y \leftarrow$ event name of the right boundary
- 3 $m1 \leftarrow$ index of the specific occurrence of event X
- 4 $m2 \leftarrow$ index of the specific occurrence of event Y
- 5 $n1 \leftarrow$ lower bound of the time distance from the $m1$ -th occurrence of event X
- 6 $n2 \leftarrow$ lower bound of the time distance from the $m2$ -th occurrence of event Y
- 7 $result \leftarrow []$
- 8 **if** $m1 = \text{null} \ \&\& \ m2 = \text{null}$ **then**
- 9 $result \leftarrow \text{applyOriginalBetweenAnd}(X, n1, Y, n2)$
- 10 **end**
- 11 **else**
- 12 **if** $m1 = \text{null}$ **then** $m1 \leftarrow 1$
- 13 **if** $m2 = \text{null}$ **then** $m2 \leftarrow 1$
- 14 $result.append(\text{applySpecialBetweenAnd}(m1, X, n1, m2, Y, n2))$
- 15 **end**

ments 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 (i1, t1)  $\leftarrow$  (0, 0)
4 (i2, t2)  $\leftarrow$  (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 i1 = 0 then
10    if e =  $X$  then
11      (i1, t1)  $\leftarrow$  (index + 1, t + n1)
12    end
13  end
14  else if e =  $Y$  && index > i1 then
15    (i2, t2)  $\leftarrow$  (index - 1, t - n2)
16    segment  $\leftarrow$  self.traceElements[i1 .. i2]
17    if n1  $\neq$  0 || n2  $\neq$  0 then
18      segment  $\leftarrow$  trace elements in segment with
19        timestamps t' satisfying t1  $\leq$  t'  $\leq$  t2
20    end
21    result.append(segment)
22    (i1, t1)  $\leftarrow$  (0, 0)
23    (i2, t2)  $\leftarrow$  (0, 0)
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 size1, size2 ← the sizes of block1 and block2
2 firstOfBlock1 ← block1.first().event
3 firstOfBlock2 ← block2.first().event
4 flag1 ← true
5 (i1, pt1) ← (1, 0)
6 (i2, pt2) ← (1, 0)
7 for elem ∈ subtrace do
8   e ← elem.event
9   t ← elem.timestamp
10  if flag1 then
11    op ← block1[i1].timeDistance.op
12    t' ← pt1 + block1[i1].timeDistance.value
13    if e = firstOfBlock1 then (i1, pt1) ← (2, t)
14    else if e = block1[i1].event && compare(t, op,
15          t') then
16      (i1, pt1) ← (i1 + 1, t)
17      if i1 = size1 + 1 then flag1 ← false
18    end
19    else (i1, pt1) ← (1, 0)
20  end
21  op ← block2[i2].timeDistance.op
22  t' ← pt2 + block2[i2].timeDistance.value
23  if e = firstOfBlock2 then
24    if flag1 || t < pt1 + n then
25      if size2 = 1 then return false
26      else (i2, pt2) ← (2, t)
27    end
28    else return true
29  end
30  else if e = block2[i2].event && compare(t, op, t')
31    then
32      if i2 = size2 then return false
33      else (i2, pt2) ← (i2 + 1, t)
34    end
35  else (i2, pt2) ← (1, 0)
36 end
37 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 defined⁹) 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 match¹⁰ 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 match¹¹ 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.

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

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

¹¹This 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 *TempPsy* properties in a tool named TEMPSPY-CHECK. The tool is based on Xtext [29] and Eclipse OCL; it is publicly available at <http://weidou.github.io/TempPsy-Check>.

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

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 available¹² 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

¹²The 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.

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.

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.

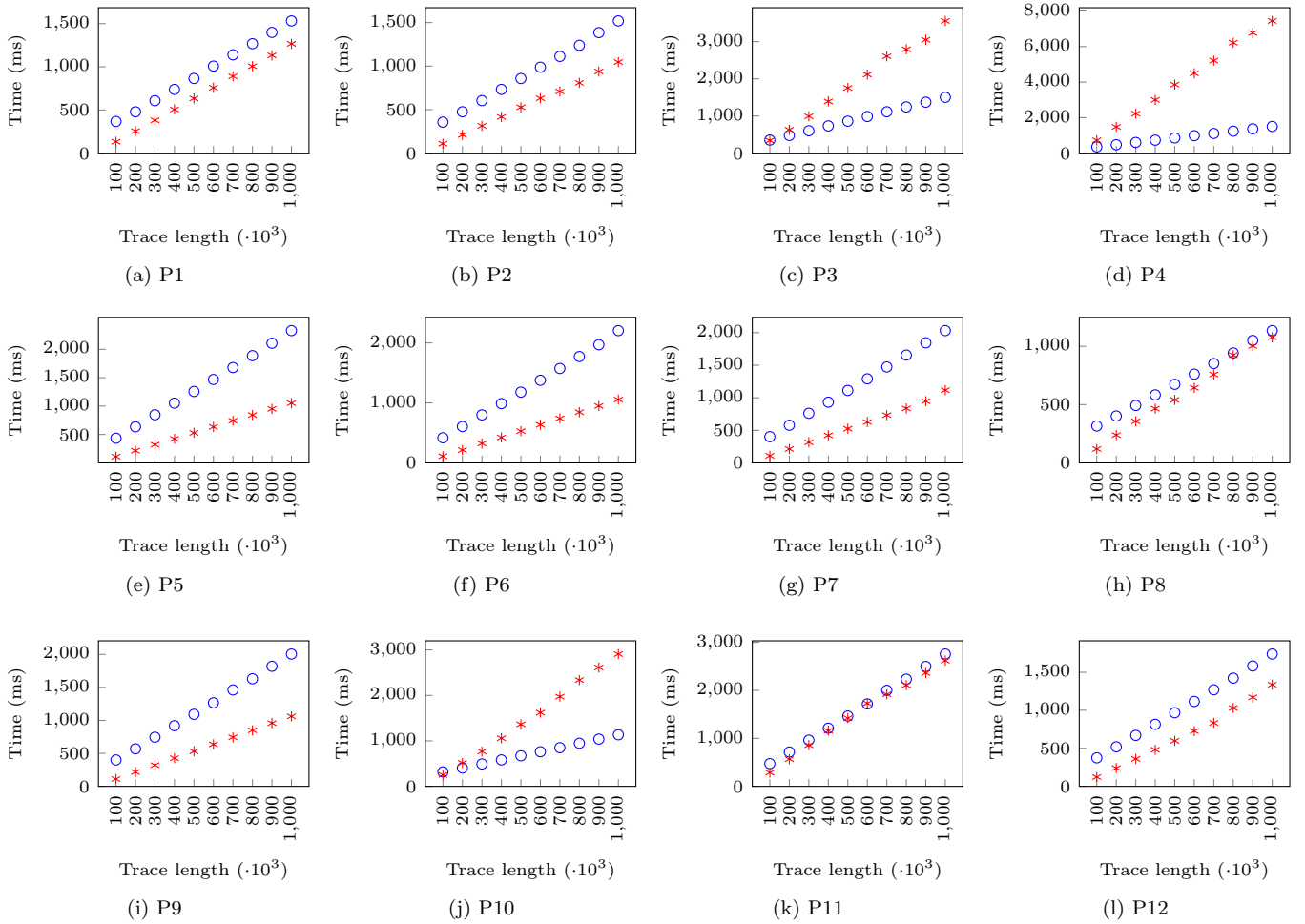


Fig. 13: Comparison between the execution time of TEMP SY-CHECK (\circ) and of MONPOLY ($*$) for properties with the *globally* scope

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 $[0, \dots, m - 1, m + 1, \dots, \text{trace length}]$.

In the case of a property containing a *precedence* or *response* pattern, we generate a number of occurrences of events (involved in the property) equal to 10% of the length of the trace. This value has been selected based on the frequency of events observed in the application whose requirements are expressed through the properties shown in Table 2. The simplest case is for a property like *globally* B *responding at most* 10 *to* 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 TEMP SY-CHECK (denoted by \circ) 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 TEMP SY-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

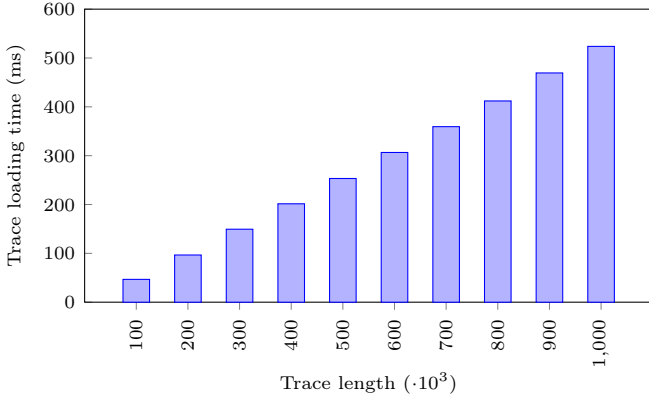


Fig. 14: Trace loading time of TEMPSY-CHECK for traces with various lengths

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 check¹³. 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.

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

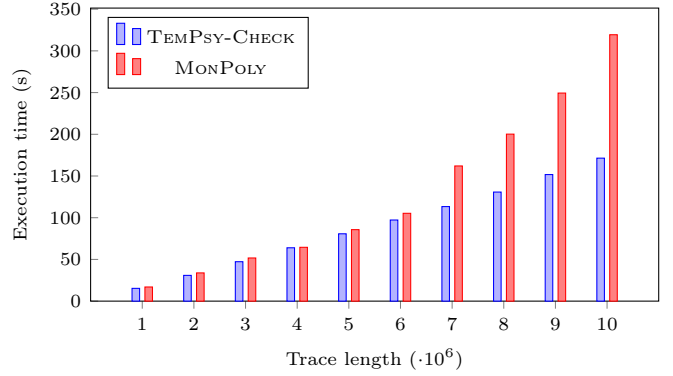


Fig. 15: Comparison of the execution time for the batch checking of ten properties with the *globally* scope

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

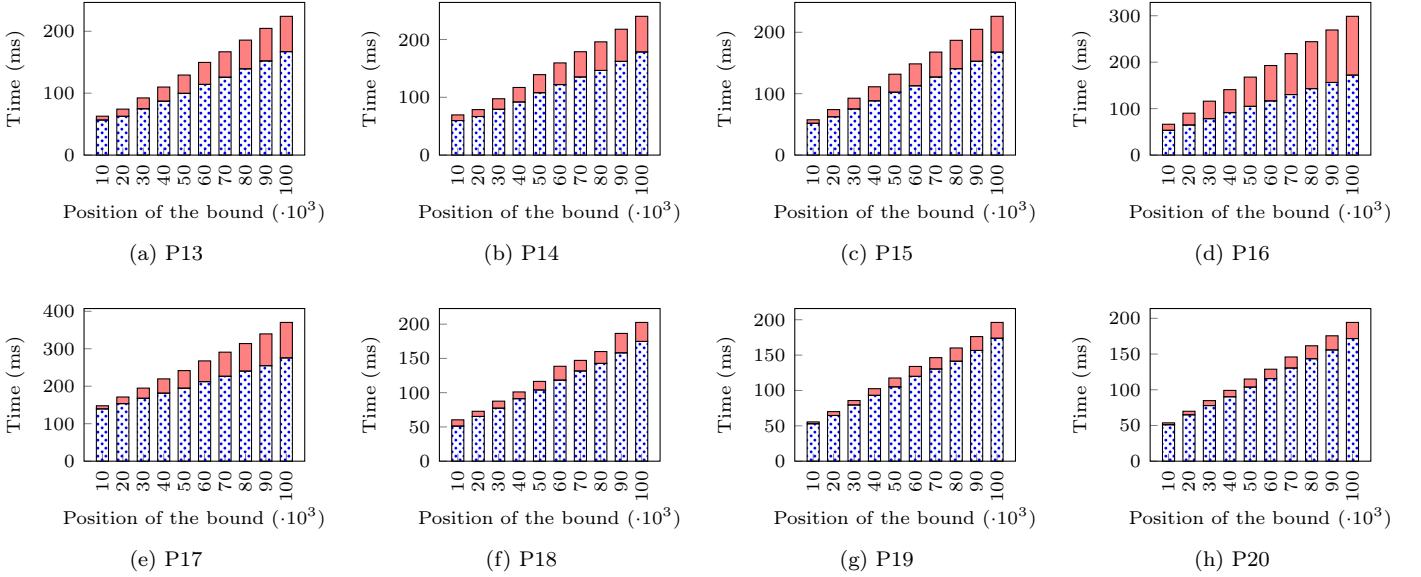


Fig. 16: Scope time \square and pattern time \blacksquare of TEMPZY-CHECK for checking properties with a *before* scope

are generated in random positions using a uniform distribution over 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 TEMPZY-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 \square) and the pattern time (denoted by \blacksquare) 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?

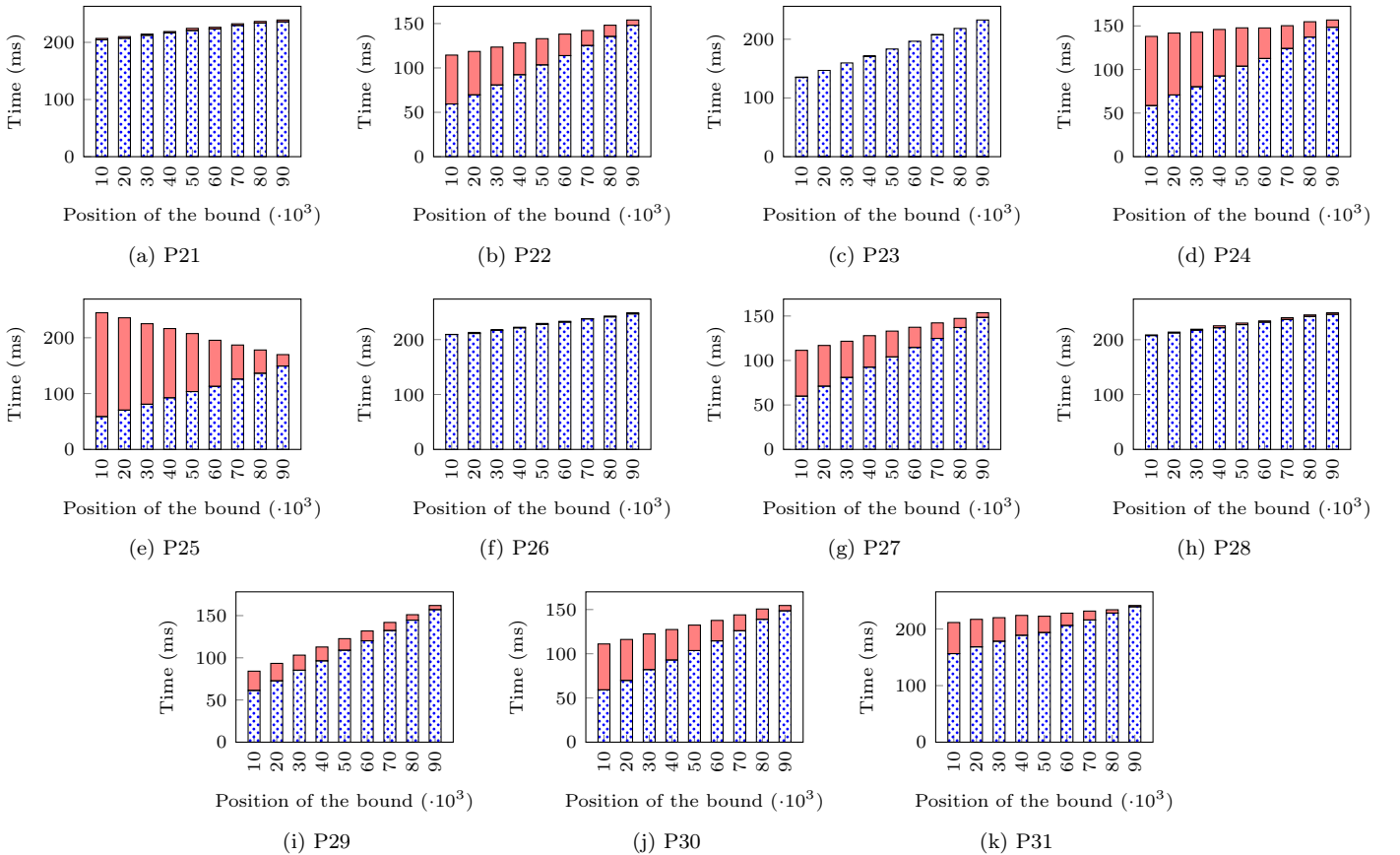




Fig. 17: Scope time  and pattern time  of TEMPSY-CHECK for checking properties with an *after* scope

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 200ms) 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

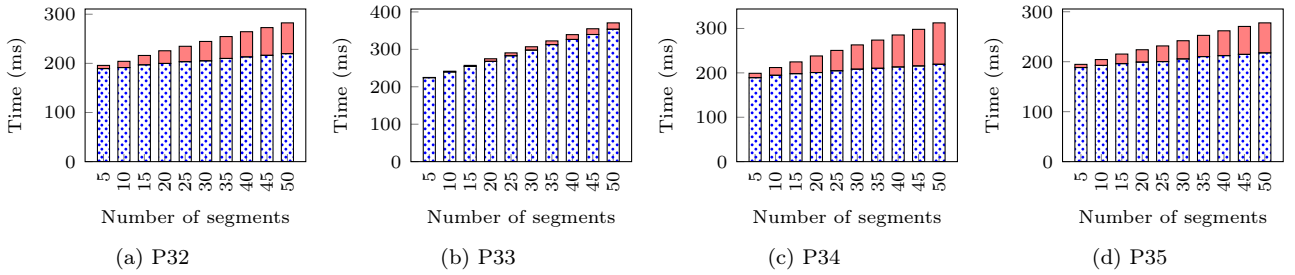


Fig. 18: Scope time ▨ and pattern time ■ of TEMP_{SY}-CHECK for checking properties with a *between-and* scope (multiple segments, fixed length)

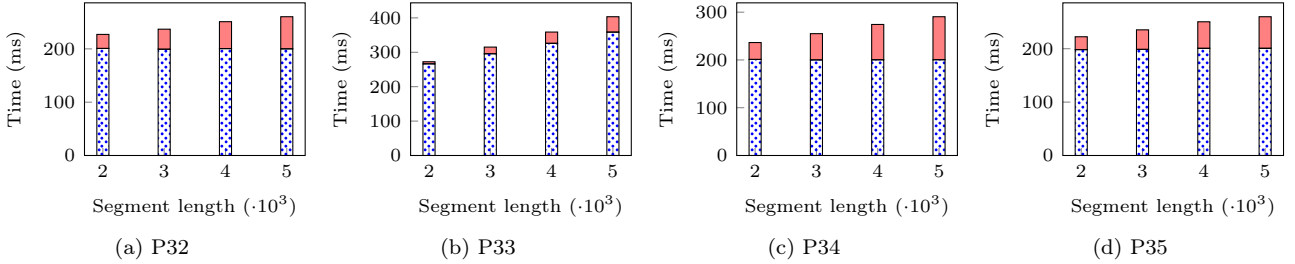


Fig. 19: Scope time ▨ and pattern time ■ of TEMP_{SY}-CHECK for checking properties with a *between-and* scope (fixed number of segments, various lengths)

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 TEMP_{SY}-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

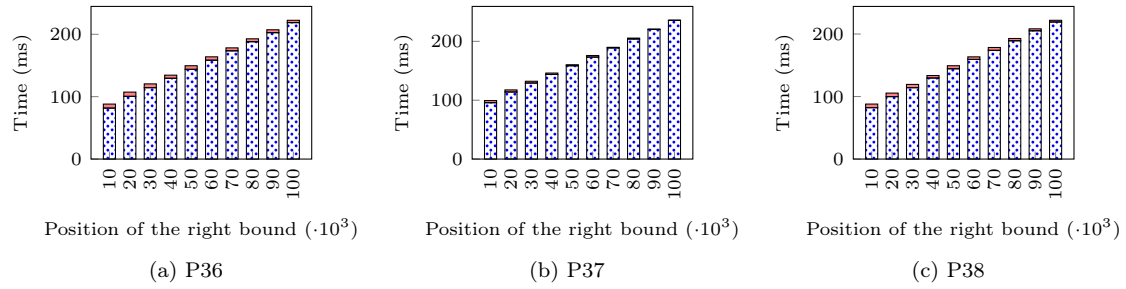


Fig. 20: Scope time ▨ and pattern time ■ of TEMPSY-CHECK for checking properties with a *between-and* scope (single segment of fixed length, different positions)

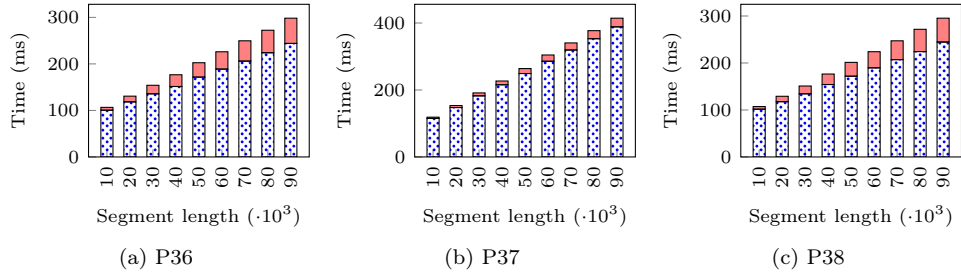


Fig. 21: Scope time ▨ and pattern time ■ of TEMPSY-CHECK for checking properties with a *between-and* scope (single segment, various lengths)

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

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

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

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.

of patterns in [27], adds also the support for specifying time distances in order patterns.

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 OCL_{TM}, developed in the context of a framework for monitoring of requirements expressed using a goal model. 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 *TempPsy*, 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, *TempPsy* 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 *TempPsy*, 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 (*TempPsy*) 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 *TempPsy* 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 *TempPsy* 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 TEMPSPY-CHECK tool, which has been made publicly available; 4) the evaluation of the scalability of TEMPSPY-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. TEMPSPY-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 verifica-

tion platform for temporal properties of business process-based applications.

In addition, as part of future work, we plan to conduct a usability study of *TempPsy*, 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 *TempPsy* 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 Éric 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 Emerich. 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

```
1 context Monitor
2
3 =====
4 def: ordinalIndexOf(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer):Integer =
5 //find the index of the n-th occurrence of the event 'eventName'
6 let result:Tuple(index:Integer, ordinal:Integer) = trace->iterate(elem:trace::TraceElement;
7   iter:Tuple(index:Integer, ordinal:Integer) = Tuple{index:Integer = 0, ordinal:Integer = 0}
8   |
9   if iter.ordinal = n then
10     iter
11   else
12     if elem.event = eventName then
13       Tuple{index:Integer = iter.index + 1, ordinal:Integer = iter.ordinal + 1}
14     else
15       Tuple{index:Integer = iter.index + 1, ordinal:Integer = iter.ordinal}
16     endif
17   endif
18 )
19 in
20 if result.ordinal = n then
21   result.index
22 else
23   -1
24 endif
25 =====
26 def: compare(a:Integer, b:Integer, which:Integer):Boolean =
27 if which = 1 then // at least b tu
28   a >= b
29 else
30   if which = 2 then // at most b tu
31     a <= b
32   else
33     if which = 3 then // exactly b tu
34       a = b
35     else
36       true // no comparison is needed
37     endif
38   endif
39 endif
40
41 =====
42 def: loadDistances(distances:Sequence(TemPsy::TimeDistance))
43       :Sequence(Tuple(which:Integer, value:Integer)) =
44 if distances->forall(elem | elem->isEmpty()) then
45   Sequence{}
46 else
47   distances->iterate(elem:TemPsy::TimeDistance;
48     iter:Sequence(Tuple(which:Integer, value:Integer)) = Sequence{}
49     |
50     if elem->isEmpty() then
51       iter->append(Tuple{which:Integer=0, value:Integer=1})
52     else
53       if TemPsy::ComparingOperator::ATLEAST = elem.comparingOperator then
54         iter->append(Tuple{which:Integer=1, value:Integer=elem.value})
55       else
56         if TemPsy::ComparingOperator::ATMOST = elem.comparingOperator then
57           iter->append(Tuple{which:Integer=2, value:Integer=elem.value})
58         else
59           iter->append(Tuple{which:Integer=3, value:Integer=elem.value})
60         endif
61       endif
62     endif
63   )
64 
```

```

61     endif
62   endif)
63 endif

```

A.2 Scopes

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

```

1  context Monitor
2
3  =====
4  def: applyScopeGlobally(trace:trace::Trace,
5                          scope:TemPsy::Scope):OrderedSet(trace::TraceElement) =
6  trace.traceElements
7
8  =====
9  def: applyScopeBefore(trace:trace::Trace, scope:TemPsy::Scope):OrderedSet(trace::TraceElement) =
10 //return the scope of 'before boundary'
11 //'boundary' : '[n] eventName [comparingOperator timeDistance tu]'
12 let boundary:TemPsy::Boundary = scope.oclAsType(TemPsy::UniScope).boundary, eventName:String = boundary.event.name in
13 if boundary.timeDistance->notEmpty() then
14   let comparingOperator:TemPsy::ComparingOperator = boundary.timeDistance.comparingOperator, timeDistance:Integer =
15     boundary.timeDistance.value in
16   if boundary.ordinal > 0 then
17     let n:Integer = boundary.ordinal in
18     if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
19       self.atLeastBefore(trace.traceElements, eventName,n,timeDistance)
20     else
21       if TemPsy::ComparingOperator::ATMOST = comparingOperator then
22         self.atMostBefore(trace.traceElements, eventName,n,timeDistance)
23       else
24         self.exactlyBefore(trace.traceElements, eventName,n,timeDistance)
25       endif
26     endif
27   else
28     if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
29       self.atLeastBefore(trace.traceElements, eventName,1,timeDistance)
30     else
31       if TemPsy::ComparingOperator::ATMOST = comparingOperator then
32         self.atMostBefore(trace.traceElements, eventName,1,timeDistance)
33       else
34         self.exactlyBefore(trace.traceElements, eventName,1,timeDistance)
35       endif
36     endif
37   else
38     if boundary.ordinal > 0 then
39       let n:Integer = boundary.ordinal in
40       self.atLeastBefore(trace.traceElements, eventName,n,1)
41     else
42       self.atLeastBefore(trace.traceElements, eventName,1,1)
43     endif
44   endif
45
46  =====
47 def: atLeastBefore(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):
48   OrderedSet(trace::TraceElement) =
49 //return the scope of 'before [n] eventName at least timeDistance tu'
50 let position:Integer = ordinalIndexOf(trace, eventName, n) in
51 if 1 <> position.abs() then
52   if 1 = timeDistance then
53     trace->subOrderedSet(1, position-1)
54   else
55     let toTimeStamp:Integer = trace->at(position).timestamp in
56     trace->select(elem | toTimeStamp - timeDistance >= elem.timestamp)
57   endif
58 else
59   OrderedSet{}
60 endif

```



```

60
61 =====
62 def: atMostBefore(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):
    OrderedSet(trace::TraceElement) =
63 //return the scope of 'before [n] eventName at most timeDistance tu'
64 let position:Integer = ordinalIndexOf(trace, eventName, n) in
65 if -1 <> position then
66     let toTimeStamp:Integer = trace->at(position).timestamp in
67     trace->select(elem | toTimeStamp - timeDistance <= elem.timestamp and toTimeStamp >= elem.timestamp)
68 else
69     OrderedSet{}
70 endif
71
72 =====
73 def: exactlyBefore(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):
    OrderedSet(trace::TraceElement) =
74 //return the scope of 'before [n] eventName exactly timeDistance tu'
75 let position:Integer = ordinalIndexOf(trace, eventName, n) in
76 if -1 <> position then
77     let toTimeStamp:Integer = trace->at(position).timestamp in
78     trace->select(elem | toTimeStamp - timeDistance = elem.timestamp)
79 else
80     OrderedSet{}
81 endif
82
83 =====
84 def: applyScopeAfter(trace:trace::Trace, scope:TemPsy::Scope):OrderedSet(trace::TraceElement) =
85 //return the scope of 'after boundary'
86 //'boundary' : '[n] eventName [comparingOperator timeDistance tu]'
87 let boundary:TemPsy::Boundary = scope.oclAsType(TemPsy::UniScope).boundary, eventName:String = boundary.event.name in
88 if boundary.timeDistance->notEmpty() then
89     let comparingOperator:TemPsy::ComparingOperator = boundary.timeDistance.comparingOperator, timeDistance:Integer =
        boundary.timeDistance.value in
90     if boundary.ordinal > 0 then
91         let n:Integer = boundary.ordinal in
92         if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
93             self.atLeastAfter(trace.traceElements, eventName, n, timeDistance)
94         else if TemPsy::ComparingOperator::ATMOST = comparingOperator then
95             self.atMostAfter(trace.traceElements, eventName, n, timeDistance)
96         else
97             self.exactlyAfter(trace.traceElements, eventName, n, timeDistance)
98         endif
99     endif
100 else
101     if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
102         self.atLeastAfter(trace.traceElements, eventName, 1, timeDistance)
103     else if TemPsy::ComparingOperator::ATMOST = comparingOperator then
104         self.atMostAfter(trace.traceElements, eventName, 1, timeDistance)
105     else
106         self.exactlyAfter(trace.traceElements, eventName, 1, timeDistance)
107     endif
108 endif
109 endif
110 else
111     if boundary.ordinal > 0 then
112         let n:Integer = boundary.ordinal in
113         self.atLeastAfter(trace.traceElements, eventName, n, 1)
114     else
115         self.atLeastAfter(trace.traceElements, eventName, 1, 1)
116     endif
117 endif
118
119 =====
120 def: atLeastAfter(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):
    OrderedSet(trace::TraceElement) =
121 //return the scope of 'after [n] eventName at least timeDistance tu'
122 let position:Integer = ordinalIndexOf(trace, eventName, n), size:Integer = trace->size() in
123 if -1 <> position and size <> position then
124     if 1 = timeDistance then

```

```

125   trace->subOrderedSet(position+1, size)
126   else
127     let fromTimeStamp:Integer = trace->at(position).timestamp in
128     trace->select(elem | fromTimeStamp + timeDistance <= elem.timestamp)
129   endif
130 else
131   OrderedSet{}
132 endif
133
134 =====
135 def: atMostAfter(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):OrderedSet
      (trace::TraceElement) =
136 //return the scope of 'after [n] eventName at most timeDistance tu'
137 let position:Integer = ordinalIndexOf(trace, eventName, n) in
138 if -1 <> position then
139   let fromTimeStamp:Integer = trace->at(position).timestamp in
140   trace->select(elem | fromTimeStamp <= elem.timestamp and fromTimeStamp + timeDistance >= elem.timestamp)
141 else
142   OrderedSet{}
143 endif
144
145 =====
146 def: exactlyAfter(trace:OrderedSet(trace::TraceElement), eventName:String, n:Integer, timeDistance:Integer):
      OrderedSet(trace::TraceElement) =
147 //return the scope of 'after [n] eventName exactly timeDistance tu'
148 let position:Integer = ordinalIndexOf(trace, eventName, n) in
149 if -1 <> position then
150   let fromTimeStamp:Integer = trace->at(position).timestamp in
151   trace->select(elem | fromTimeStamp + timeDistance = elem.timestamp)
152 else
153   OrderedSet{}
154 endif
155
156 =====
157
158 def: applyScopeBetweenAnd(trace:trace::Trace,
159                           scope:TemPsy::Scope
160                           :OrderedSet(OrderedSet(trace::TraceElement)) =
161 // return the scope of 'between boundaryBegin and boundaryEnd'
162 // i.e., 'between [nBegin] eventNameBegin [at least timeDistanceBegin]
163 // and [nEnd] eventNameEnd [at least timeDistanceEnd]'
164 let boundaryBegin:TemPsy::Boundary
165   = scope.oClAsType(TemPsy::BiScope).boundaryBegin,
166   boundaryEnd:TemPsy::Boundary
167   = scope.oClAsType(TemPsy::BiScope).boundaryEnd,
168   eventNameBegin:String
169   = boundaryBegin.event.name,
170   eventNameEnd:String
171   = boundaryEnd.event.name
172 in
173 if boundaryBegin.timeDistance->notEmpty() then
174   let timeDistanceBegin:Integer = boundaryBegin.timeDistance.value in
175   if boundaryEnd.timeDistance->notEmpty() then
176     let timeDistanceEnd:Integer = boundaryEnd.timeDistance.value in
177     if boundaryBegin.ordinal > 0 then
178       let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
179       nBegin:Integer = boundaryBegin.ordinal
180     in
181     if boundaryEnd.ordinal > 0 then
182       let nEnd:Integer = boundaryEnd.ordinal in
183       result->append(
184         self.applySpecialBetweenAnd(trace.traceElements,
185                                     eventNameBegin, nBegin, timeDistanceBegin,
186                                     eventNameEnd, nEnd, timeDistanceEnd))
187     else
188       result->append(
189         self.applySpecialBetweenAnd(trace.traceElements,
190                                     eventNameBegin, nBegin, timeDistanceBegin,
191                                     eventNameEnd, 1, timeDistanceEnd))

```

```

192     endif
193 else
194     if boundaryEnd.ordinal > 0 then
195         let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
196             nEnd:Integer = boundaryEnd.ordinal
197         in
198             result->append(
199                 self.applySpecialBetweenAnd(trace.traceElements,
200                     eventNameBegin, 1, timeDistanceBegin,
201                     eventNameEnd, nEnd, timeDistanceEnd))
202     else
203         self.applyOriginalBetweenAnd(trace.traceElements,
204             eventNameBegin, timeDistanceBegin,
205             eventNameEnd, timeDistanceEnd)
206     endif
207 endif
208 else
209     if boundaryBegin.ordinal > 0 then
210         let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
211             nBegin:Integer = boundaryBegin.ordinal
212         in
213             if boundaryEnd.ordinal > 0 then
214                 let nEnd:Integer = boundaryEnd.ordinal in
215                     result->append(
216                         self.applySpecialBetweenAnd(trace.traceElements,
217                             eventNameBegin, nBegin, timeDistanceBegin,
218                             eventNameEnd, nEnd, 1))
219             else
220                 result->append(
221                     self.applySpecialBetweenAnd(trace.traceElements,
222                         eventNameBegin, nBegin, timeDistanceBegin,
223                         eventNameEnd, 1, 1))
224             endif
225         else
226             if boundaryEnd.ordinal > 0 then
227                 let result:OrderedSet(OrderedSet(trace::TraceElement))
228                     = OrderedSet{},
229                     nEnd:Integer = boundaryEnd.ordinal
230                 in
231                     result->append(
232                         self.applySpecialBetweenAnd(trace.traceElements,
233                             eventNameBegin, 1, timeDistanceBegin,
234                             eventNameEnd, nEnd, 1))
235             else
236                 self.applyOriginalBetweenAnd(trace.traceElements,
237                     eventNameBegin, timeDistanceBegin, eventNameEnd)
238             endif
239         endif
240     endif
241 else
242     if boundaryEnd.timeDistance->notEmpty() then
243         let timeDistanceEnd:Integer = boundaryEnd.timeDistance.value in
244             if boundaryBegin.ordinal > 0 then
245                 let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
246                     nBegin:Integer = boundaryBegin.ordinal
247                 in
248                     if boundaryEnd.ordinal > 0 then
249                         let nEnd:Integer = boundaryEnd.ordinal in
250                             result->append(
251                                 self.applySpecialBetweenAnd(trace.traceElements,
252                                     eventNameBegin, nBegin, 1, eventNameEnd,
253                                     nEnd, timeDistanceEnd))
254                     else
255                         result->append(
256                             self.applySpecialBetweenAnd(trace.traceElements,
257                                 eventNameBegin, nBegin, 1,
258                                 eventNameEnd, 1, timeDistanceEnd))
259                     endif
260             else

```

```

261     if boundaryEnd.ordinal > 0 then
262         let result:OrderedSet(OrderedSet(trace::TraceElement))
263             = OrderedSet{},
264             nEnd:Integer = boundaryEnd.ordinal
265         in
266             result->append(
267                 self.applySpecialBetweenAnd(trace.traceElements,
268                     eventNameBegin, 1, 1,
269                     eventNameEnd, nEnd, timeDistanceEnd))
270     else
271         self.applyOriginalBetweenAnd(trace.traceElements,
272             eventNameBegin, eventNameEnd, timeDistanceEnd)
273     endif
274 endif
275 else
276 if boundaryBegin.ordinal > 0 then
277     let result:OrderedSet(OrderedSet(trace::TraceElement)) = OrderedSet{},
278         nBegin:Integer = boundaryBegin.ordinal
279     in
280     if boundaryEnd.ordinal > 0 then
281         let nEnd:Integer = boundaryEnd.ordinal in
282             result->append(
283                 self.applySpecialBetweenAnd(trace.traceElements,
284                     eventNameBegin, nBegin, 1,
285                     eventNameEnd, nEnd, 1))
286     else
287         result->append(
288             self.applySpecialBetweenAnd(trace.traceElements,
289                 eventNameBegin, nBegin, 1,
290                 eventNameEnd, 1, 1))
291     endif
292 else
293     if boundaryEnd.ordinal > 0 then
294         let result:OrderedSet(OrderedSet(trace::TraceElement))
295             = OrderedSet{},
296             nEnd:Integer = boundaryEnd.ordinal
297         in
298             result->append(
299                 self.applySpecialBetweenAnd(trace.traceElements,
300                     eventNameBegin, 1, 1,
301                     eventNameEnd, nEnd, 1))
302     else
303         self.applyOriginalBetweenAnd(trace.traceElements, eventNameBegin, eventNameEnd)
304     endif
305 endif
306 endif
307 endif
308
309 =====
310 def: applyOriginalBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, eventNameEnd:String):
311     Sequence(OrderedSet(trace::TraceElement)) =
312 //return the scope of 'between eventNameBegin and eventNameEnd'
313 trace->iterate(elem:trace::TraceElement;
314     iter:Tuple(index:Integer, result:Sequence(OrderedSet(trace::TraceElement)), i:Integer)
315     =Tuple{index:Integer = 0, result:Sequence(OrderedSet(trace::TraceElement)) = Sequence{}, i:Integer = 0} |
316     if iter.i = 0 then
317         let currentIndex:Integer = iter.index + 1 in
318         if elem.event = eventNameBegin then
319             Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
320                 currentIndex}
321         else
322             Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
323                 iter.i}
324         endif
325     endif
326     else
327         if elem.event = eventNameEnd then
328             let i:Integer = iter.i+1, j:Integer = iter.index in
329             if i <= j then
330                 Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result->append(trace->

```

```

        subOrderedSet(i, j)), i:Integer = 0}
327     else
328         Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = 0}
329     endif
330     else
331         Tuple{index:Integer = iter.index + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer
            = iter.i}
332     endif
333     endif
334 ).result
335
336 =====
337 def: applyOriginalBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, distanceBegin:Integer,
        eventNameEnd:String):Sequence(OrderedSet(trace::TraceElement)) =
338 //return the scope of 'between eventNameBegin at least distanceBegin tu and eventNameEnd'
339 trace->iterate(elem:trace::TraceElement;
340     iter:Tuple(index:Integer, result:Sequence(OrderedSet(trace::TraceElement)), i:Integer, criticalTime:Integer)
341     =Tuple{index:Integer = 0, result:Sequence(OrderedSet(trace::TraceElement)) = Sequence{}, i:Integer = 0,
        criticalTime:Integer = 0} |
342     let e:String = elem.event in
343     if iter.i = 0 then
344         let currentIndex:Integer = iter.index + 1 in
345         if e = eventNameBegin then
346             Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
                currentIndex, criticalTime:Integer = elem.timestamp + distanceBegin}
347         else
348             Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
                iter.i, criticalTime:Integer = iter.criticalTime}
349         endif
350     else
351         if e = eventNameEnd then
352             let t:Integer = elem.timestamp, i:Integer = iter.i + 1, j:Integer = iter.index, t1:Integer = iter.criticalTime
                in
353             if i <= j and t1 < t then
354                 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}
355             else
356                 Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = 0,
                    criticalTime:Integer = iter.criticalTime}
357             endif
358         else
359             Tuple{index:Integer = iter.index + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer
                = iter.i, criticalTime:Integer = iter.criticalTime}
360         endif
361     endif
362 ).result
363
364 =====
365 def: applyOriginalBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, eventNameEnd:String,
        distanceEnd:Integer):Sequence(OrderedSet(trace::TraceElement)) =
366 //return the scope of 'between eventNameBegin and at least distanceEnd tu eventNameEnd'
367 trace->iterate(elem:trace::TraceElement;
368     iter:Tuple(index:Integer, result:Sequence(OrderedSet(trace::TraceElement)), i:Integer, criticalTime:Integer)
369     =Tuple{index:Integer = 0, result:Sequence(OrderedSet(trace::TraceElement)) = Sequence{}, i:Integer = 0,
        criticalTime:Integer = 0} |
370     let e:String = elem.event in
371     if iter.i = 0 then
372         let currentIndex:Integer = iter.index + 1 in
373         if e = eventNameBegin then
374             Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
                currentIndex, criticalTime:Integer = elem.timestamp + 1}
375         else
376             Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
                iter.i, criticalTime:Integer = iter.criticalTime}
377         endif
378     else
379         if e = eventNameEnd then
380             let t:Integer = elem.timestamp, i:Integer = iter.i + 1, j:Integer = iter.index, t1:Integer = iter.criticalTime,

```

```

        t2:Integer = t - distanceEnd in
381   if i <= j and t1 <= t2 then
382     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}
383   else
384     Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = 0,
        criticalTime:Integer = iter.criticalTime}
385   endif
386   else
387     Tuple{index:Integer = iter.index + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer
        = iter.i, criticalTime:Integer = iter.criticalTime}
388   endif
389   endif
390 ).result
391
392 =====
393 def: applyOriginalBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, distanceBegin:Integer,
        eventNameEnd:String, distanceEnd:Integer):Sequence(OrderedSet(trace::TraceElement)) =
394 //return the scope of 'between eventNameBegin at least distanceBegin tu and at least distanceEnd tu eventNameEnd'
395 trace->iterate(elem:trace::TraceElement;
396   iter:Tuple(index:Integer, result:Sequence(OrderedSet(trace::TraceElement)), i:Integer, criticalTime:Integer)
397   =Tuple{index:Integer = 0, result:Sequence(OrderedSet(trace::TraceElement)) = Sequence{}, i:Integer = 0,
        criticalTime:Integer = 0} |
398   let e:String = elem.event in
399   if iter.i = 0 then
400     let currentIndex:Integer = iter.index + 1 in
401     if e = eventNameBegin then
402       Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
        currentIndex, criticalTime:Integer = elem.timestamp + distanceBegin}
403     else
404       Tuple{index:Integer = currentIndex, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer =
        iter.i, criticalTime:Integer = iter.criticalTime}
405     endif
406   else
407     if e = eventNameEnd then
408       let t:Integer = elem.timestamp, i:Integer = iter.i + 1, j:Integer = iter.index, t1:Integer = iter.criticalTime,
        t2:Integer = t - distanceEnd in
409       if i <= j and t1 <= t2 then
410         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}
411       else
412         Tuple{index:Integer = j + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer = 0,
        criticalTime:Integer = iter.criticalTime}
413       endif
414     else
415       Tuple{index:Integer = iter.index + 1, result:Sequence(OrderedSet(trace::TraceElement)) = iter.result, i:Integer
        = iter.i, criticalTime:Integer = iter.criticalTime}
416     endif
417   endif
418 ).result
419
420 =====
421 def: applySpecialBetweenAnd(trace:OrderedSet(trace::TraceElement), eventNameBegin:String, nBegin:Integer,
        timeDistanceBegin:Integer, eventNameEnd:String, nEnd:Integer, timeDistanceEnd:Integer):OrderedSet(trace::
        TraceElement) =
422 //return the scope of 'between nBegin eventNameBegin at least timeDistanceBegin tu and nBegin eventNameEnd at least
        timeDistanceEnd tu'
423 let t:Tuple(index:Integer, indexBegin:Integer, indexEnd:Integer, count:Integer) = trace->iterate(elem:trace::
        TraceElement;
424   iter:Tuple(index:Integer, indexBegin:Integer, indexEnd:Integer, count:Integer) = Tuple{index:Integer = 0,
        indexBegin:Integer = 0, indexEnd:Integer = 0, count:Integer = 0} |
425   if iter.indexBegin = 0 then
426     let currentIndex:Integer = iter.index + 1 in
427     if elem.event = eventNameBegin then
428       let currentBeginCount:Integer = iter.count+1 in
429       if currentBeginCount = nBegin then
430         Tuple{index:Integer = currentIndex, indexBegin:Integer = currentIndex + 1, indexEnd:Integer = iter.indexEnd

```

```

        , count:Integer = 0}
431     else
432         Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = iter.indexEnd,
            count:Integer = currentBeginCount}
433     endif
434     else
435         Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = iter.indexEnd,
            count:Integer = iter.count}
436     endif
437     else
438         if iter.indexEnd = 0 then
439             let currentIndex:Integer = iter.index + 1 in
440                 if elem.event = eventNameEnd then
441                     let currentEndCount:Integer = iter.count+1 in
442                         if currentEndCount = nEnd then
443                             Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = currentIndex
                                -1, count:Integer = nEnd}
444                         else
445                             Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = iter.
                                indexEnd, count:Integer = currentEndCount}
446                         endif
447                     else
448                         Tuple{index:Integer = currentIndex, indexBegin:Integer = iter.indexBegin, indexEnd:Integer = iter.indexEnd,
                                count:Integer = iter.count}
449                     endif
450                 else
451                     iter
452                 endif
453             endif
454         )
455     in
456     let
457         i:Integer = t.indexBegin,
458         j:Integer = t.indexEnd,
459         timestampBegin:Integer = trace->at(i-1).timestamp+timeDistanceBegin,
460         timestampEnd:Integer = trace->at(j+1).timestamp-timeDistanceEnd
461     in
462     if i > 0 and j > 0 and i <= j then
463         if timeDistanceBegin = 1 and timeDistanceEnd = 1 then
464             trace->subOrderedSet(i, j)
465         else
466             trace->subOrderedSet(i, j)->select(elem | elem.timestamp >= timestampBegin and elem.timestamp <= timestampEnd)
467         endif
468     else
469         OrderedSet{}
470     endif

```

A.3 Patterns

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

```

1  context Monitor
2
3  =====
4  def: checkPatternUniversality(subtrace:OrderedSet(trace::TraceElement), pattern:TemPsy::Pattern):Boolean =
5  // check the satisfiability of the universality pattern 'always eventName'
6  let eventName:String = pattern.oclAsType(TemPsy::Universality).event.name in
7  subtrace->forall(event = eventName)
8
9  =====
10 def: checkPatternExistence(subtrace:OrderedSet(trace::TraceElement), pattern:TemPsy::Pattern):Boolean =
11 --check the satisfiability of the existence pattern 'pattern'
12 if subtrace->isEmpty() then
13     true
14 else
15     let occPattern:TemPsy::OccurrencePattern = pattern.oclAsType(TemPsy::OccurrencePattern), eventName:String =
        occPattern.event.name in
16     if occPattern.comparingOperator->notEmpty() then

```

```

17   let comparingOperator:TemPsy::ComparingOperator = occPattern.comparingOperator, n:Integer = occPattern.times,
      count:Integer = subtrace.event->count(eventName) in
18   if TemPsy::ComparingOperator::ATLEAST = comparingOperator then
19     count >= n
20   else
21     if TemPsy::ComparingOperator::ATMOST = comparingOperator then
22       count <= n
23     else
24       count = n
25     endif
26   endif
27   else
28     subtrace.event->includes(eventName)
29   endif
30 endif
31
32 =====
33 def: checkPatternAbsence(subtrace:OrderedSet(trace::TraceElement), pattern:TemPsy::Pattern):Boolean =
34 --check the satisfiability of the absence pattern 'pattern'
35 if subtrace->isEmpty() then
36   true
37 else
38   let occPattern:TemPsy::OccurrencePattern = pattern.oclAsType(TemPsy::OccurrencePattern), eventName:String =
      occPattern.event.name in
39   if occPattern.comparingOperator->notEmpty() then
40     subtrace.event->count(eventName) <> occPattern.times
41   else
42     subtrace.event->excludes(eventName)
43   endif
44 endif
45
46 =====
47     else
48       self.checkPatternPrecedenceOneManyRight(subtrace, cause,
49         effects, effectDistances)
50     endif
51   else
52     self.checkPatternPrecedenceManyManyRight(subtrace, causes,
53       effects, effectDistances)
54   endif
55   else
56     if causeSize = 1 then
57       let cause:String=causes->first() in
58       if effectSize = 2 then
59         let effectDistance:Tuple(which:Integer, value:Integer)
60           =effectDistances->first()
61         in
62         self.checkPatternPrecedenceOneTwoMidRight(subtrace, cause,
63           orderPattern.timeDistance,
64           effects->first(),
65           effectDistance, effects->at(2))
66       else
67         let distance:Tuple(which:Integer, value:Integer)
68           =self.loadDistance(orderPattern.timeDistance)
69         in
70         self.checkPatternPrecedenceOneManyMidRight(subtrace, cause,
71           distance, effects,
72           effectDistances)
73       endif
74     else
75       let distance:Tuple(which:Integer, value:Integer)
76         =self.loadDistance(orderPattern.timeDistance)
77       in
78       self.checkPatternPrecedenceManyManyMidRight(subtrace, causes,
79         distance, effects, effectDistances)
80     endif
81   endif
82 endif
83 else

```



```

84  if effectDistances->isEmpty() then
85    if orderPattern.timeDistance->isEmpty() then
86      if effectSize = 1 then
87        let effect:String=effects->first() in
88        if causeSize = 2 then
89          let causeDistance:Tuple(which:Integer, value:Integer)
90            =causeDistances->first()
91          in
92            self.checkPatternPrecedenceTwoOneLeft(subtrace, causes->first(),
93              causeDistance, causes->at(2), effect)
94        else
95          self.checkPatternPrecedenceManyOneLeft(subtrace, causes,
96            causeDistances, effect)
97        endif
98      else
99        self.checkPatternPrecedenceManyManyLeft(subtrace, causes,
100          causeDistances, effects)
101      endif
102    else
103      let distance:Tuple(which:Integer, value:Integer)
104        =self.loadDistance(orderPattern.timeDistance)
105      in
106      if effectSize = 1 then
107        let effect:String=effects->first() in
108        self.checkPatternPrecedenceManyOneLeftMid(subtrace, causes,
109          causeDistances, distance, effect)
110      else
111        self.checkPatternPrecedenceManyManyLeftMid(subtrace, causes,
112          causeDistances, distance, effects)
113      endif
114    endif
115  else
116    if orderPattern.timeDistance->isEmpty() then
117      self.checkPatternPrecedenceManyManyLeftRight(subtrace, causes, causeDistances,
118        effects, effectDistances)
119    else
120      let distance:Tuple(which:Integer, value:Integer)
121        =self.loadDistance(orderPattern.timeDistance)
122      in
123      self.checkPatternPrecedenceManyManyLeftMidRight(subtrace, causes, causeDistances,
124        distance, effects, effectDistances)
125    endif
126  endif
127 endif
128 endif
129
130 =====
131 def: checkPatternPrecedenceOneOnePlain(subtrace:OrderedSet(trace::TraceElement), cause:String, effect:String):Boolean
132   =
133   /"cause preceding effect"
134   subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, result:Integer) = Tuple{flag:Boolean = true,
135     result:Integer = 0}
136   |
137   if iter.flag then
138     let e:String = elem.event in
139     if e = cause then
140       Tuple{flag:Boolean = false, result:Integer = -1}
141     else
142       if e = effect then
143         Tuple{flag:Boolean = false, result:Integer = -2} // violation
144       else
145         iter
146       endif
147     endif
148   else
149     iter
150   endif
151 ).result >= -1

```

```

151 =====
152 def: checkPatternPrecedenceOneOneAtLeastMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effect:String):Boolean =
153 /"cause preceding at least distance tu effect"
154 subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstant:Integer) = Tuple(flag:Boolean
      = true, midCriticalInstant:Integer = 0)
155 |
156 if iter.flag then
157   let e:String = elem.event in
158   if iter.midCriticalInstant = 0 and e = cause then //catch the first occurrence of cause
159     Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance)
160   else
161     if e = effect then
162       if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
163         Tuple(flag:Boolean = false, midCriticalInstant:Integer = -2) // violation
164       else
165         Tuple(flag:Boolean = false, midCriticalInstant:Integer = -1)
166       endif
167     else
168       iter
169     endif
170   endif
171 else
172   iter
173 endif
174 ).midCriticalInstant >= -1
175
176 =====
177 def: checkPatternPrecedenceOneOneAtMostMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effect:String):Boolean =
178 /"cause preceding at most distance tu effect"
179 subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstant:Integer) = Tuple(flag:Boolean
      = true, midCriticalInstant:Integer = 0)
180 |
181 if iter.flag then
182   let e:String = elem.event in
183   if e = cause then //latest cause
184     Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance)
185   else
186     if e = effect and elem.timestamp > iter.midCriticalInstant then
187       Tuple(flag:Boolean = false, midCriticalInstant:Integer = null) // violation
188     else
189       iter
190     endif
191   endif
192 else
193   iter
194 endif
195 ).flag
196
197 =====
198 def: checkPatternPrecedenceOneOneExactlyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effect:String):Boolean =
199 subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer)) = Tuple{
      flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}} |
200 if iter.flag then
201   let e:String = elem.event in
202   if e = cause then
203     Tuple(flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.
      timestamp+distance))
204   else
205     if e = effect then
206       let t:Integer = elem.timestamp in
207       if iter.midCriticalInstants->includes(t) then
208         Tuple(flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(
      subElem | subElem > t))
209     else
210       Tuple(flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null)
211     endif

```

```

212     else
213         iter
214     endif
215 endif
216 else
217     iter
218 endif
219 ).flag
220
221 =====
222 def: checkPatternPrecedenceOneOneMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:TemPsy::
    TimeDistance, effect:String):Boolean =
223 let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
224 if which = TemPsy::ComparingOperator::ATLEAST then
225     self.checkPatternPrecedenceOneOneAtLeastMid(subtrace, cause, value, effect)
226 else
227     if which = TemPsy::ComparingOperator::ATMOST then
228         self.checkPatternPrecedenceOneOneAtMostMid(subtrace, cause, value, effect)
229     else
230         self.checkPatternPrecedenceOneOneExactlyMid(subtrace, cause, value, effect)
231     endif
232 endif
233
234 =====
235 def: checkPatternPrecedenceOneManyPlain(subtrace:OrderedSet(trace::TraceElement), cause:String, effects:Sequence(
    String)):Boolean =
236 let
237     effectSize:Integer = effects->size(),
238     firstEffect:String = effects->first()
239 in
240 subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, result:Integer, i2:Integer) = Tuple{flag:Boolean
    = true, result:Integer = 0, i2:Integer = 1}
241 |
242 if iter.flag then
243     let e:String = elem.event in
244     if e = cause then //catch the first occurrence of cause
245         Tuple{flag:Boolean = false, result:Integer = -1, i2:Integer = null}
246     else
247         if e = effects->at(iter.i2) then
248             if iter.i2 = effectSize then
249                 Tuple{flag:Boolean = false, result:Integer = -2, i2:Integer = null}
250             else
251                 Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = iter.i2 + 1}
252             endif
253         else
254             if e = firstEffect then
255                 Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = 2}
256             else
257                 Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = 1}
258             endif
259         endif
260     endif
261 else
262     iter
263 endif
264 ).result >= -1
265
266 =====
267 def: checkPatternPrecedenceOneManyAtLeastMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer
    , effects:Sequence(String)):Boolean =
268 let
269     effectSize:Integer = effects->size(),
270     firstEffect:String = effects->first()
271 in
272 subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer) = Tuple{
    flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1}
273 |
274 if iter.flag then
275     let e:String = elem.event in

```

```

276 if iter.midCriticalInstant = 0 and e = cause then //catch the first occurrence of cause
277   Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i2:Integer = 1}
278 else
279   if iter.i2 > 1 and e = effects->at(iter.i2) then
280     if iter.i2 = effectSize then
281       Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i2:Integer = null}
282     else
283       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2
284         + 1}
285     endif
286   else
287     if e = firstEffect then
288       if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
289         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
290       else
291         Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i2:Integer = null}
292       endif
293     else
294       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
295     endif
296   endif
297 else
298   iter
299 endif
300 ).midCriticalInstant >= -1
301
302 =====
303 def: checkPatternPrecedenceOneManyAtMostMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
304   effects:Sequence(String)):Boolean =
305 let
306   effectSize:Integer = effects->size(),
307   firstEffect:String = effects->first()
308 in
309   subtrace->iterate(elem:trace::TraceElement;
310     iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:
311       Integer = 0, i2:Integer = 1} |
312     let e:String = elem.event in
313     if iter.flag then
314       if e = cause then
315         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i2:Integer = 1}
316       else
317         if iter.i2 > 1 and e = effects->at(iter.i2) then
318           if iter.i2 = effectSize then
319             Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i2:Integer = null}
320           else
321             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2
322               + 1}
323           endif
324         else
325           if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
326             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
327           else
328             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
329           endif
330         endif
331       endif
332     else
333       iter
334     endif
335   ).flag
336 =====
337 def: checkPatternPrecedenceOneManyExactlyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer
338   , effects:Sequence(String)):Boolean =
339 let
340   effectSize:Integer = effects->size(),
341   firstEffect:String = effects->first()
342 in

```

```

340 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}
341 |
342 if iter.flag then
343   let e:String = elem.event in
344   if e = cause then
345     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.
      timestamp+distance), i2:Integer = 1}
346   else
347     if iter.i2 > 1 and e = effects->at(iter.i2) then
348       if iter.i2 = effectSize then
349         Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i2:Integer = null}
350       else
351         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:
          Integer = iter.i2 + 1}
352     endif
353   else
354     if e = firstEffect then
355       let t:Integer = elem.timestamp in
356       if iter.midCriticalInstants->includes(t) then
357         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(
          subElem | subElem > t), i2:Integer = 1}
358       else
359         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:
          Integer = 2}
360     endif
361   else
362     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:
      Integer = 1}
363   endif
364 endif
365 endif
366 else
367   iter
368 endif
369 ).flag
370
371 =====
372 def: checkPatternPrecedenceOneManyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:TempPsy::
      TimeDistance, effects:Sequence(String)):Boolean =
373 let value:Integer = distance.value, which:TempPsy::ComparingOperator = distance.comparingOperator in
374 if which = TempPsy::ComparingOperator::ATLEAST then
375   self.checkPatternPrecedenceOneManyAtLeastMid(subtrace, cause, value, effects)
376 else
377   if which = TempPsy::ComparingOperator::ATMOST then
378     self.checkPatternPrecedenceOneManyAtMostMid(subtrace, cause, value, effects)
379   else
380     self.checkPatternPrecedenceOneManyExactlyMid(subtrace, cause, value, effects)
381   endif
382 endif
383
384
385 =====
386 def: checkPatternPrecedenceOneManyRight(subtrace:OrderedSet(trace::TraceElement), cause:String, effects:Sequence(
      String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
387 let
388   effectSize:Integer = effects->size(),
389   firstEffect:String = effects->first(),
390   secondEffectDistance:Integer = effectDistances->at(2).value
391 in
392 subtrace->iterate(elem:trace::TraceElement;
393   iter:Tuple(flag:Boolean, result:Integer, i2:Integer, effectCriticalInstant:Integer)
394   = Tuple{flag:Boolean = true, result:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
395   |
396   if iter.flag then
397     let e:String = elem.event in
398     if e = cause then
399       Tuple{flag:Boolean = false, result:Integer = -1, i2:Integer = null, effectCriticalInstant:Integer = null}
400     else

```

```

401   if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
      effectDistances->at(iter.i2).which) then
402     if iter.i2 = effectSize then
403       Tuple{flag:Boolean = false, result:Integer = -2, i2:Integer = null, effectCriticalInstant:Integer = null}
404     else
405       let i22:Integer = iter.i2 + 1 in
406       Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = i22, effectCriticalInstant:
          Integer = elem.timestamp + effectDistances->at(i22).value}
407     endif
408   else
409     if e = firstEffect then
410       Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = 2, effectCriticalInstant:Integer
          = elem.timestamp + secondEffectDistance}
411     else
412       Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i2:Integer = 1, effectCriticalInstant:Integer
          = iter.effectCriticalInstant}
413     endif
414   endif
415 endif
416 else
417   iter
418 endif
419 ).result >= -1
420
421 =====
422 def: checkPatternPrecedenceOneManyAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance
      :Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
423 let
424   effectSize:Integer = effects->size(),
425   firstEffect:String = effects->first(),
426   secondEffectDistance:Integer = effectDistances->at(2).value
427 in
428 subtrace->iterate(elem:trace::TraceElement;
429   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
430   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
431   |
432   if iter.flag then
433     let e:String = elem.event in
434     if iter.midCriticalInstant = 0 and e = cause then //catch the first occurrence of cause
435       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i2:Integer = 1,
          effectCriticalInstant:Integer = iter.effectCriticalInstant}
436     else
437       if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
          effectDistances->at(iter.i2).which) then
438         if iter.i2 = effectSize then
439           Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i2:Integer = null, effectCriticalInstant:
              Integer = null}
440         else
441           let i22:Integer = iter.i2 + 1 in
442           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i22,
              effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
443         endif
444       else
445         if e = firstEffect then
446           if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
447             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2,
              effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
448           else
449             Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i2:Integer = null, effectCriticalInstant:
              Integer = null}
450           endif
451         else
452           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1,
              effectCriticalInstant:Integer = iter.effectCriticalInstant}
453         endif
454       endif
455     endif
456   else
457     iter

```

```

458 endif
459 ).midCriticalInstant >= -1
460
461 =====
462 def: checkPatternPrecedenceOneManyAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:
      Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
463 let
464   effectSize:Integer = effects->size(),
465   firstEffect:String = effects->first(),
466   secondEffectDistance:Integer = effectDistances->at(2).value
467 in
468 subtrace->iterate(elem:trace::TraceElement;
469   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
470 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
471 |
472 if iter.flag then
473   let e:String = elem.event in
474   if e = cause then
475     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i2:Integer = 1,
      effectCriticalInstant:Integer = iter.effectCriticalInstant}
476   else
477     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
      effectDistances->at(iter.i2).which) then
478       if iter.i2 = effectSize then
479         Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:
          Integer = null}
480       else
481         let i22:Integer = iter.i2 + 1 in
482         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i22,
          effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
483       endif
484     else
485       if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
486         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2,
          effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
487       else
488         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1,
          effectCriticalInstant:Integer = iter.effectCriticalInstant}
489       endif
490     endif
491   endif
492   else
493     iter
494   endif
495 ).flag
496
497 // added on 18/08/2015
498 def: checkPatternPrecedenceOneManyExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance
      :Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
499 let
500   effectSize:Integer = effects->size(),
501   firstEffect:String = effects->first(),
502   secondEffectDistance:Integer = effectDistances->at(2).value
503 in
504 subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i2:
      Integer, effectCriticalInstant:Integer)
505 = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i2:Integer = 1,
      effectCriticalInstant:Integer = 0}
506 |
507 if iter.flag then
508   let e:String = elem.event in
509   if e = cause then
510     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.
      timestamp+midDistance), i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
511   else
512     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
      effectDistances->at(iter.i2).which) then
513       if iter.i2 = effectSize then
514         Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i2:Integer = null,

```

```

        effectCriticalInstant:Integer = null}
515     else
516         let i22:Integer = iter.i2 + 1 in
517         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:
            Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
518     endif
519     else
520         if e = firstEffect then
521             let t:Integer = elem.timestamp in
522             if iter.midCriticalInstants->includes(t) then
523                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(
                    subElem | subElem > t), i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
524             else
525                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:
                    Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
526             endif
527         else
528             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i2:
                Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
529         endif
530     endif
531     endif
532     else
533         iter
534     endif
535 ).flag
536
537 =====
538 def: checkPatternPrecedenceOneManyMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:TemPsy
    ::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean
    =
539 let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
540 if midWhich = TemPsy::ComparingOperator::ATLEAST then
541     self.checkPatternPrecedenceOneManyAtLeastMidRight(subtrace, cause, midValue, effects, effectDistances)
542 else
543     if midWhich = TemPsy::ComparingOperator::ATMOST then
544         self.checkPatternPrecedenceOneManyAtMostMidRight(subtrace, cause, midValue, effects, effectDistances)
545     else
546         self.checkPatternPrecedenceOneManyExactlyMidRight(subtrace, cause, midValue, effects, effectDistances)
547     endif
548 endif
549
550 =====
551 def: checkPatternPrecedenceManyOnePlain(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effect:
    String):Boolean =
552 let
553     causeSize:Integer = causes->size(),
554     firstCause:String = causes->first()
555 in
556 subtrace->iterate(elem:trace::TraceElement; iter:Tuple(flag:Boolean, result:Integer, i1:Integer) = Tuple(flag:Boolean
    = true, result:Integer = 0, i1:Integer = 1}
557 |
558 if iter.flag then
559     let e:String = elem.event in
560     if iter.i1 > 1 and e = causes->at(iter.i1) then
561         if iter.i1 = causeSize then
562             Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null}
563         else
564             Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1}
565         endif
566     else
567         if e = firstCause then
568             Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2}
569         else
570             if e = effect then
571                 Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null}
572             else
573                 Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1}
574             endif

```



```

575     endif
576   endif
577   else
578     iter
579   endif
580 ).result >= -1
581
582 =====
583 def: checkPatternPrecedenceManyOneAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
584     distance:Integer, effect:String):Boolean =
585 let
586   causeSize:Integer = causes->size(),
587   firstCause:String = causes->first()
588 in
589 subtrace->iterate(elem:trace::TraceElement;
590   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:
591     Integer = 0, i1:Integer = 1}
592 |
593 if iter.flag then
594   let e:String = elem.event in
595   if iter.i1 > 1 and e = causes->at(iter.i1) then
596     if iter.i1 = causeSize then
597       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1}
598     else
599       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 +
600         1}
601     endif
602   else
603     if iter.midCriticalInstant = 0 and e = firstCause then
604       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
605     else
606       if e = effect then
607         if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
608           Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null}
609         else
610           Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null}
611         endif
612       else
613         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
614       endif
615     endif
616   endif
617 else
618   iter
619 endif
620 ).midCriticalInstant >= -1
621
622 =====
623 def: checkPatternPrecedenceManyOneAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
624     distance:Integer, effect:String):Boolean =
625 let
626   causeSize:Integer = causes->size(),
627   firstCause:String = causes->first()
628 in
629 subtrace->iterate(elem:trace::TraceElement;
630   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer) = Tuple{flag:Boolean = true, midCriticalInstant:
631     Integer = 0, i1:Integer = 1}
632 |
633 if iter.flag then
634   let e:String = elem.event in
635   if iter.i1 > 1 and e = causes->at(iter.i1) then
636     if iter.i1 = causeSize then
637       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1}
638     else
639       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 +
640         1}
641     endif
642   else
643     if e = firstCause then
644       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
645     else
646       if e = effect then
647         if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
648           Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null}
649         else
650           Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null}
651         endif
652       else
653         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
654       endif
655     endif
656   endif
657 else
658   iter
659 endif
660 ).midCriticalInstant >= -1

```

```

638     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
639   else
640     if e = effect and elem.timestamp > iter.midCriticalInstant then
641       Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null}
642     else
643       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
644     endif
645   endif
646 endif
647 else
648   iter
649 endif
650 ).flag
651
652 =====
653 def: checkPatternPrecedenceManyOneExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
        distance:Integer, effect:String):Boolean =
654 let
655   causeSize:Integer = causes->size(),
656   firstCause:String = causes->first()
657 in
658 subtrace->iterate(elem:trace::TraceElement;
659   iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer)
660   = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1}
661   |
662   if iter.flag then
663     let e:String = elem.event in
664     if iter.i1 > 1 and e = causes->at(iter.i1) then
665       if iter.i1 = causeSize then
666         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem
        .timestamp+distance), i1:Integer = 1}
667       else
668         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer
        = iter.i1 + 1}
669       endif
670     else
671       if e = firstCause then
672         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer
        = 2}
673       else
674         if e = effect then
675           let t:Integer = elem.timestamp in
676           if iter.midCriticalInstants->includes(t) then
677             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(
        subElem | subElem > t), i1:Integer = 1}
678           else
679             Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null}
680           endif
681         else
682           Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
        Integer = 1}
683         endif
684       endif
685     endif
686   else
687     iter
688   endif
689 ).flag
690
691 =====
692 def: checkPatternPrecedenceManyOneMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:
        TemPsy::TimeDistance, effect:String):Boolean =
693 let value:Integer = distance.value, which:TemPsy::ComparingOperator=distance.comparingOperator in
694 if which = TemPsy::ComparingOperator::ATLEAST then
695   self.checkPatternPrecedenceManyOneAtLeastMid(subtrace, causes, value, effect)
696 else
697   if which = TemPsy::ComparingOperator::ATMOST then
698     self.checkPatternPrecedenceManyOneAtMostMid(subtrace, causes, value, effect)
699   else

```

```

700 self.checkPatternPrecedenceManyOneExactlyMid(subtrace, causes, value, effect)
701 endif
702 endif
703
704 =====
705 def: checkPatternPrecedenceManyOneLeft(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effect:String):Boolean =
706 let
707   causeSize:Integer = causes->size(),
708   firstCause:String = causes->first(),
709   secondCauseDistance:Integer = causeDistances->at(2).value
710 in
711 subtrace->iterate(elem:trace::TraceElement;
712   iter:Tuple(flag:Boolean, result:Integer, i1:Integer, causeCriticalInstant:Integer)
713   = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
714   |
715   if iter.flag then
716     let e:String = elem.event in
717     if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
        causeDistances->at(iter.i1).which) then
718       if iter.i1 = causeSize then
719         Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null}
720       else
721         let i11:Integer = iter.i1 + 1 in
722         Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:Integer
          = elem.timestamp + causeDistances->at(i11).value}
723       endif
724     else
725       if e = firstCause then
726         Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:Integer =
          elem.timestamp + secondCauseDistance}
727       else
728         if e = effect then
729           Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer = null}
730         else
731           Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:Integer
            = iter.causeCriticalInstant}
732         endif
733       endif
734     endif
735   else
736     iter
737   endif
738 ).result >= -1
739
740 =====
741 def: checkPatternPrecedenceManyOneLeftAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
742 let
743   causeSize:Integer = causes->size(),
744   firstCause:String = causes->first(),
745   secondCauseDistance:Integer = causeDistances->at(2).value
746 in
747 subtrace->iterate(elem:trace::TraceElement;
748   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer)
749   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
750   |
751   if iter.flag then
752     let e:String = elem.event in
753     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
754       if iter.i1 = causeSize then
755         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
          causeCriticalInstant:Integer = iter.causeCriticalInstant}
756       else
757         let i11:Integer = iter.i1 + 1 in
758         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
          causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value}
759       endif

```

```

760 else
761   if iter.midCriticalInstant = 0 and e = firstCause then
762     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
763           causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance}
764   else
765     if e = effect then
766       if iter.midCriticalInstant = 0 or elem.timestamp < iter.midCriticalInstant then
767         Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, causeCriticalInstant:
768               Integer = null}
769       else
770         Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:
771               Integer = null}
772       endif
773     else
774       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
775             causeCriticalInstant:Integer = iter.causeCriticalInstant}
776     endif
777   endif
778   iter
779   .midCriticalInstant >= -1
780   =====
781 def: checkPatternPrecedenceManyOneLeftAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
782         causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
783 let
784   causeSize:Integer = causes->size(),
785   firstCause:String = causes->first(),
786   secondCauseDistance:Integer = causeDistances->at(2).value
787 in
788   subtrace->iterate(elem:trace::TraceElement;
789     iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer)
790     = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
791     |
792     if iter.flag then
793       let e:String = elem.event in
794       if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
795             causeDistances->at(iter.i1).which) then
796         if iter.i1 = causeSize then
797           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
798                 causeCriticalInstant:Integer = iter.causeCriticalInstant}
799         else
800           let i11:Integer = iter.i1 + 1 in
801           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
802                 causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value}
803         endif
804       else
805         if e = firstCause then
806           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
807                 causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance}
808         else
809           if e = effect and elem.timestamp > iter.midCriticalInstant then
810             Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:
811                   Integer = null}
812           else
813             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
814                   causeCriticalInstant:Integer = iter.causeCriticalInstant}
815           endif
816         endif
817       endif
818     else
819       iter
820     endif
821   ).flag
822   =====
823 def: checkPatternPrecedenceManyOneLeftExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),

```

```

      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
818 let
819   causeSize:Integer = causes->size(),
820   firstCause:String = causes->first(),
821   secondCauseDistance:Integer = causeDistances->at(2).value
822 in
823 subtrace->iterate(elem:trace::TraceElement;
824   iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, causeCriticalInstant:Integer)
825   = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1,
      causeCriticalInstant:Integer = 0}
826   |
827   if iter.flag then
828     let e:String = elem.event in
829     if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
      causeDistances->at(iter.i1).which) then
830       if iter.i1 = causeSize then
831         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem
          .timestamp+midDistance), i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
832       else
833         let i1l:Integer = iter.i1 + 1 in
834         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer
          = i1l, causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i1l).value}
835       endif
836     else
837       if e = firstCause then
838         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:Integer
          = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance}
839       else
840         if e = effect then
841           let t:Integer = elem.timestamp in
842           if iter.midCriticalInstants->includes(t) then
843             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->select(
              subElem | subElem > t), i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
844           else
845             Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null,
              causeCriticalInstant:Integer = null}
846           endif
847         else
848           Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
            Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
849         endif
850       endif
851     endif
852   else
853     iter
854   endif
855 ).flag
856
857 =====
858 def: checkPatternPrecedenceManyOneLeftMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TempPsy::TimeDistance, effect:String):
      Boolean =
859 let midValue:Integer = midDistance.value, midWhich:TempPsy::ComparingOperator=midDistance.comparingOperator in
860 if midWhich = TempPsy::ComparingOperator::ATLEAST then
861   self.checkPatternPrecedenceManyOneLeftAtLeastMid(subtrace, causes, causeDistances, midValue, effect)
862 else
863   if midWhich = TempPsy::ComparingOperator::ATMOST then
864     self.checkPatternPrecedenceManyOneLeftAtMostMid(subtrace, causes, causeDistances, midValue, effect)
865   else
866     self.checkPatternPrecedenceManyOneLeftExactlyMid(subtrace, causes, causeDistances, midValue, effect)
867   endif
868 endif
869
870 =====
871 def: checkPatternPrecedenceManyManyPlain(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effects:
      Sequence(String)):Boolean =
872 let
873   causeSize:Integer = causes->size(),
874   firstCause:String = causes->first(),

```

```

875 effectSize:Integer = effects->size(),
876 firstEffect:String = effects->first(),
877 lastEffect:String = effects->last()
878 in
879 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}
880 |
881 if iter.flag then
882   let e:String = elem.event in
883   if iter.i2 = effectSize and e = lastEffect then
884     Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, i2:Integer = null}
885   else
886     if iter.i1 > 1 and e = causes->at(iter.i1) then
887       if iter.i1 = causeSize then
888         Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, i2:Integer = null}
889       else
890         if e = effects->at(iter.i2) then
891           Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = iter
             .i2 + 1}
892         else
893           if e = firstEffect then
894             Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = 2}
895           else
896             Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = iter.i1 + 1, i2:Integer = 1}
897           endif
898         endif
899       endif
900     else
901       if e = firstCause then
902         if e = effects->at(iter.i2) then
903           Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = iter.i2 + 1}
904         else
905           if e = firstEffect then
906             Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = 2}
907           else
908             Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, i2:Integer = 1}
909           endif
910         endif
911       else
912         if e = effects->at(iter.i2) then
913           Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = iter.i2 + 1}
914         else
915           if e = firstEffect then
916             Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = 2}
917           else
918             Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, i2:Integer = 1}
919           endif
920         endif
921       endif
922     endif
923   endif
924 else
925   iter
926 endif
927 ).result >= -1
928
929 =====
930 def: checkPatternPrecedenceManyManyAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      distance:Integer, effects:Sequence(String)):Boolean =
931 let
932   causeSize:Integer = causes->size(),
933   firstCause:String = causes->first(),
934   effectSize:Integer = effects->size(),
935   firstEffect:String = effects->first(),
936   lastEffect:String = effects->last()
937 in
938 subtrace->iterate(elem:trace::TraceElement;
939   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer) = Tuple{flag:Boolean = true,
      midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1} |

```

```

940 if iter.flag then
941   let e:String = elem.event in
942   if iter.midCriticalInstant > 0 and elem.timestamp >= iter.midCriticalInstant then
943     Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, i2:Integer = null} //
       satisfaction
944   else
945     if iter.i2 = effectSize and e = lastEffect then
946       Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, i2:Integer = null} //
       violation
947     else
948       if iter.i1 > 1 and e = causes->at(iter.i1) then
949         if iter.i1 = causeSize then
950           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1,
             i2:Integer = 1}
951         else
952           if e = effects->at(iter.i2) then
953             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
954           else
955             if e = firstEffect then
956               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
                 iter.i1 + 1, i2:Integer = 2} // a potential violation
957             else
958               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
                 iter.i1 + 1, i2:Integer = 1}
959             endif
960           endif
961         endif
962       else
963         if iter.midCriticalInstant = 0 and e = firstCause then
964           if e = effects->at(iter.i2) then
965             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
               i2:Integer = iter.i2 + 1}
966           else
967             if e = firstEffect then
968               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                 i2:Integer = 2} // a potential violation
969             else
970               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                 i2:Integer = 1}
971             endif
972           endif
973         else
974           if e = effects->at(iter.i2) then
975             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
               i2:Integer = iter.i2 + 1}
976           else
977             if e = firstEffect then
978               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                 i2:Integer = 2} // a potential violation
979             else
980               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                 i2:Integer = 1}
981             endif
982           endif
983         endif
984       endif
985     endif
986   endif
987 else
988   iter
989 endif
990 ).midCriticalInstant >= -1
991
992 =====
993 def: checkPatternPrecedenceManyManyAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
       distance:Integer, effects:Sequence(String)):Boolean =
994 let
995   causeSize:Integer = causes->size(),

```

```

996 firstCause:String = causes->first(),
997 effectSize:Integer = effects->size(),
998 firstEffect:String = effects->first(),
999 lastEffect:String = effects->last()
1000 in
1001 subtrace->iterate(elem:trace::TraceElement;
1002   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer) = Tuple(flag:Boolean = true,
      midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1)
1003   |
1004   if iter.flag then
1005     let e:String = elem.event in
1006     if iter.i2 = effectSize and e = lastEffect then
1007       Tuple(flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null)
1008     else
1009       if iter.i1 > 1 and e = causes->at(iter.i1) then
1010         if iter.i1 = causeSize then
1011           Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1, i2:
             Integer = 1)
1012         else
1013           if iter.i2 > 1 and e = effects->at(iter.i2) then
1014             Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
               i1 + 1, i2:Integer = iter.i2 + 1)
1015           else
1016             if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1017               Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
                 .i1 + 1, i2:Integer = 2)
1018             else
1019               Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
                 .i1 + 1, i2:Integer = 1)
1020             endif
1021           endif
1022         endif
1023       else
1024         if e = firstCause then
1025           if iter.i2 > 1 and e = effects->at(iter.i2) then
1026             Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
               Integer = iter.i2 + 1)
1027           else
1028             if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1029               Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                 i2:Integer = 2)
1030             else
1031               Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                 i2:Integer = 1)
1032             endif
1033           endif
1034         else
1035           if iter.i2 > 1 and e = effects->at(iter.i2) then
1036             Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
               Integer = iter.i2 + 1)
1037           else
1038             if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1039               Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                 i2:Integer = 2)
1040             else
1041               Tuple(flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                 i2:Integer = 1)
1042             endif
1043           endif
1044         endif
1045       endif
1046     endif
1047   else
1048     iter
1049   endif
1050 ).flag
1051
1052 =====
1053 def: checkPatternPrecedenceManyManyExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),

```



```

distance:Integer, effects:Sequence(String)):Boolean =
1054 let
1055   causeSize:Integer = causes->size(),
1056   firstCause:String = causes->first(),
1057   effectSize:Integer = effects->size(),
1058   firstEffect:String = effects->first(),
1059   lastEffect:String = effects->last()
1060 in
1061   subtrace->iterate(elem:trace::TraceElement;
1062     iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, i2:Integer)
1063     = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1, i2:Integer = 1}
1064     |
1065     if iter.flag then
1066       let e:String = elem.event in
1067       if iter.i2 = effectSize and e = lastEffect then
1068         Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null, i2:Integer = null}
1069       else
1070         if iter.i1 > 1 and e = causes->at(iter.i1) then
1071           if iter.i1 = causeSize then
1072             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(
1073               elem.timestamp+distance), i1:Integer = 1, i2:Integer = 1}
1074           else
1075             if iter.i2 > 1 and e = effects->at(iter.i2) then
1076               Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
1077                 Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1}
1078             else
1079               if e = firstEffect then
1080                 let t:Integer = elem.timestamp in
1081                 if iter.midCriticalInstants->includes(t) then
1082                   Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
1083                     select(subElem | subElem > t), i1:Integer = iter.i1 + 1, i2:Integer = 1}
1084                 else
1085                   Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
1086                     Integer = iter.i1 + 1, i2:Integer = 2}
1087                 endif
1088             else
1089               Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
1090                 Integer = iter.i1 + 1, i2:Integer = 1}
1091             endif
1092           endif
1093         else
1094           if e = firstCause then
1095             if iter.i2 > 1 and e = effects->at(iter.i2) then
1096               Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
1097                 Integer = 2, i2:Integer = iter.i2 + 1}
1098             else
1099               if e = firstEffect then
1100                 let t:Integer = elem.timestamp in
1101                 if iter.midCriticalInstants->includes(t) then
1102                   Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
1103                     select(subElem | subElem > t), i1:Integer = 2, i2:Integer = 1}
1104                 else
1105                   Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
1106                     Integer = 2, i2:Integer = 2}
1107                 endif
1108             else
1109               Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
1110                 Integer = 2, i2:Integer = 1}
1111             endif
1112         endif
1113       endif
1114     endif
1115   else
1116     if iter.i2 > 1 and e = effects->at(iter.i2) then
1117       Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
1118         Integer = 1, i2:Integer = iter.i2 + 1}
1119     else
1120       if e = firstEffect then
1121         let t:Integer = elem.timestamp in
1122         if iter.midCriticalInstants->includes(t) then

```

```

1112         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
           select(subElem | subElem > t), i1:Integer = 1, i2:Integer = 1}
1113     else
1114         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
           Integer = 1, i2:Integer = 2}
1115     endif
1116 else
1117     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
           Integer = 1, i2:Integer = 1}
1118 endif
1119 endif
1120 endif
1121 endif
1122 endif
1123 else
1124     iter
1125 endif
1126 ).flag
1127
1128 =====
1129 def: checkPatternPrecedenceManyManyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:
      TempPsy::TimeDistance, effects:Sequence(String)):Boolean =
1130 let value:Integer = distance.value, which:TempPsy::ComparingOperator=distance.comparingOperator in
1131 if which = TempPsy::ComparingOperator::ATLEAST then
1132     self.checkPatternPrecedenceManyManyAtLeastMid(subtrace, causes, value, effects)
1133 else
1134     if which = TempPsy::ComparingOperator::ATMOST then
1135         self.checkPatternPrecedenceManyManyAtMostMid(subtrace, causes, value, effects)
1136     else
1137         self.checkPatternPrecedenceManyManyExactlyMid(subtrace, causes, value, effects)
1138     endif
1139 endif
1140
1141 =====
1142 def: checkPatternPrecedenceManyManyLeft(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effects:Sequence(String)):Boolean =
1143 let
1144     causeSize:Integer = causes->size(),
1145     firstCause:String = causes->first(),
1146     secondCauseDistance:Integer = causeDistances->at(2).value,
1147     effectSize:Integer = effects->size(),
1148     firstEffect:String = effects->first(),
1149     lastEffect:String = effects->last()
1150 in
1151 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}
1152 |
1153 if iter.flag then
1154     let e:String = elem.event in
1155     if iter.i2 = effectSize and e = lastEffect then
1156         Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
           Integer = null}
1157     else
1158         if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
           causeDistances->at(iter.i1).which) then
1159             if iter.i1 = causeSize then
1160                 Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null, i2
                   :Integer = null}
1161             else
1162                 let i11:Integer = iter.i1 + 1 in
1163                 if e = effects->at(iter.i2) then
1164                     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}
1165                 else
1166                     if e = firstEffect then
1167                         Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:
                           Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = 2}
1168                     else

```

```

1169         Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = i11, causeCriticalInstant:
           Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = 1}
1170     endif
1171     endif
1172     endif
1173     else
1174         if e = firstCause then
1175             if e = effects->at(iter.i2) then
1176                 Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:
                   Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
1177             else
1178                 if e = firstEffect then
1179                     Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:
                       Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
1180                 else
1181                     Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:
                       Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
1182                 endif
1183             endif
1184         else
1185             if e = effects->at(iter.i2) then
1186                 Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:
                   Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
1187             else
1188                 if e = firstEffect then
1189                     Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:
                       Integer = iter.causeCriticalInstant, i2:Integer = 2}
1190                 else
1191                     Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:
                       Integer = iter.causeCriticalInstant, i2:Integer = 1}
1192                 endif
1193             endif
1194         endif
1195     endif
1196     endif
1197 else
1198     iter
1199     endif
1200 ).result >= -1
1201
1202 =====
1203 def: checkPatternPrecedenceManyManyLeftAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
           causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)):
           Boolean =
1204 let
1205     causeSize:Integer = causes->size(),
1206     firstCause:String = causes->first(),
1207     secondCauseDistance:Integer = causeDistances->at(2).value,
1208     effectSize:Integer = effects->size(),
1209     firstEffect:String = effects->first(),
1210     lastEffect:String = effects->last()
1211 in
1212 subtrace->iterate(elem:trace::TraceElement;
1213     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}
1214 |
1215 if iter.flag then
1216     let e:String = elem.event in
1217     if iter.midCriticalInstant > 0 and elem.timestamp >= iter.midCriticalInstant then
1218         Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer =
           null, i2:Integer = null} // satisfaction
1219     else
1220         if iter.i2 = effectSize and e = lastEffect then
1221             Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer
               = null, i2:Integer = null} // violation
1222         else
1223             if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
               causeDistances->at(iter.i1).which) then

```

```

1224     if iter.i1 = causeSize then
1225         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer =
1226             1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
1227     else
1228         let i1:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11
1229             ).value in
1230         if e = effects->at(iter.i2) then // for instance {causes: [a,b,c], effects: [d,a,b]}, when i1 = 1, i2 = 2
1231             or i1 = 2, i2 = 3. But it is not possible i1 equals to causeSize, since causes cannot be a sublist
1232             of effects.
1233             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
1234                 causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = iter.i2 + 1}
1235         else
1236             if e = firstEffect then
1237                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
1238                     i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2} // a potential
1239                     violation
1240             else
1241                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
1242                     i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1}
1243             endif
1244         endif
1245     endif
1246     else
1247         if iter.midCriticalInstant = 0 and e = firstCause then
1248             if e = effects->at(iter.i2) then
1249                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
1250                     causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
1251             else
1252                 if e = firstEffect then
1253                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
1254                         causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2} // a
1255                         potential violation
1256                 else
1257                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
1258                         causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
1259                 endif
1260             endif
1261         else
1262             if e = effects->at(iter.i2) then
1263                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
1264                     causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
1265             else
1266                 if e = firstEffect then
1267                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
1268                         causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2} // a potential
1269                         violation
1270                 else
1271                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
1272                         causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
1273                 endif
1274             endif
1275         endif
1276     endif
1277     else
1278         iter
1279     endif
1280 ).midCriticalInstant >= -1
1281
1282 =====
1283 def: checkPatternPrecedenceManyManyLeftAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
1284     causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)):
1285     Boolean =
1286 let
1287     causeSize:Integer = causes->size(),
1288     firstCause:String = causes->first(),
1289     secondCauseDistance:Integer = causeDistances->at(2).value,
1290     effectSize:Integer = effects->size(),

```

```

1275 firstEffect:String = effects->first(),
1276 lastEffect:String = effects->last()
1277 in
1278 subtrace->iterate(elem:trace::TraceElement;
1279 iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer)
1280 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
Integer = 1}
1281 |
1282 if iter.flag then
1283 let e:String = elem.event in
1284 if iter.i2 = effectSize and e = lastEffect then
1285 Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer
= null, i2:Integer = null}
1286 else
1287 if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
causeDistances->at(iter.i1).which) then
1288 if iter.i1 = causeSize then
1289 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
1290 else
1291 let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).
value in
1292 if iter.i2 > 1 and e = effects->at(iter.i2) then
1293 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = iter.i2 + 1}
1294 else
1295 if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1296 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2}
1297 else
1298 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1}
1299 endif
1300 endif
1301 endif
1302 else
1303 if e = firstCause then
1304 if iter.i2 > 1 and e = effects->at(iter.i2) then
1305 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
1306 else
1307 if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1308 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
1309 else
1310 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
1311 endif
1312 endif
1313 else
1314 if iter.i2 > 1 and e = effects->at(iter.i2) then
1315 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
1316 else
1317 if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1318 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
1319 else
1320 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
1321 endif
1322 endif
1323 endif
1324 endif
1325 endif
1326 else
1327 iter
1328 endif
1329 ).flag

```

```

1330
1331 =====
1332 def: checkPatternPrecedenceManyManyLeftExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)):
      Boolean =
1333 let
1334   causeSize:Integer = causes->size(),
1335   firstCause:String = causes->first(),
1336   secondCauseDistance:Integer = causeDistances->at(2).value,
1337   effectSize:Integer = effects->size(),
1338   firstEffect:String = effects->first(),
1339   lastEffect:String = effects->last()
1340 in
1341 subtrace->iterate(elem:trace::TraceElement;
1342   iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, causeCriticalInstant:Integer, i2:
      Integer)
1343   = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1,
      causeCriticalInstant:Integer = 0, i2:Integer = 1}
1344   |
1345   if iter.flag then
1346     let e:String = elem.event in
1347     if iter.i2 = effectSize and e = lastEffect then
1348       Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null,
        causeCriticalInstant:Integer = null, i2:Integer = null}
1349   else
1350     if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
        causeDistances->at(iter.i1).which) then
1351       if iter.i1 = causeSize then
1352         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(
          elem.timestamp+midDistance), i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
          i2:Integer = 1}
1353     else
1354       let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).
        value in
1355       if iter.i2 > 1 and e = effects->at(iter.i2) then
1356         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
          Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = iter.i2 + 1}
1357     else
1358       if e = firstEffect then
1359         let t:Integer = elem.timestamp in
1360         if iter.midCriticalInstants->includes(t) then
1361           Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
            select(subElem | subElem > t), i1:Integer = i11, causeCriticalInstant:Integer =
            nextCauseCriticalInstant, i2:Integer = 1}
1362         else
1363           Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
            Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2}
1364       endif
1365     else
1366       Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
        Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1}
1367     endif
1368   endif
1369   endif
1370   else
1371     if e = firstCause then
1372       if iter.i2 > 1 and e = effects->at(iter.i2) then
1373         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
          Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.
          i2 + 1}
1374     else
1375       if e = firstEffect then
1376         let t:Integer = elem.timestamp in
1377         if iter.midCriticalInstants->includes(t) then
1378           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}
1379         else
1380           Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:

```

```

Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer =
2}
1381     endif
1382     else
1383         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
Integer = 2, causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
1384     endif
1385     endif
1386     else
1387         if iter.i2 > 1 and e = effects->at(iter.i2) then
1388             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
1389         else
1390             if e = firstEffect then
1391                 let t:Integer = elem.timestamp in
1392                 if iter.midCriticalInstants->includes(t) then
1393                     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
select(subElem | subElem > t), i1:Integer = 1, causeCriticalInstant:Integer = iter.
causeCriticalInstant, i2:Integer = 1}
1394                 else
1395                     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
1396                 endif
1397             else
1398                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
1399             endif
1400         endif
1401     endif
1402     endif
1403     endif
1404     else
1405         iter
1406     endif
1407 ).flag
1408
1409 =====
1410 def: checkPatternPrecedenceManyManyLeftMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TemPsy::TimeDistance, effects:Sequence
(String)):Boolean =
1411 let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
1412 if midWhich = TemPsy::ComparingOperator::ATLEAST then
1413     self.checkPatternPrecedenceManyManyLeftAtLeastMid(subtrace, causes, causeDistances, midValue, effects)
1414 else
1415     if midWhich = TemPsy::ComparingOperator::ATMOST then
1416         self.checkPatternPrecedenceManyManyLeftAtMostMid(subtrace, causes, causeDistances, midValue, effects)
1417     else
1418         self.checkPatternPrecedenceManyManyLeftExactlyMid(subtrace, causes, causeDistances, midValue, effects)
1419     endif
1420 endif
1421
1422 =====
1423 def: checkPatternPrecedenceManyManyRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effects:
Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
1424 let
1425     causeSize:Integer = causes->size(),
1426     firstCause:String = causes->first(),
1427     effectSize:Integer = effects->size(),
1428     firstEffect:String = effects->first(),
1429     lastEffect:String = effects->last(),
1430     secondEffectDistance:Integer = effectDistances->at(2).value
1431 in
1432 subtrace->iterate(elem:trace::TraceElement;
1433 iter:Tuple(flag:Boolean, result:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
1434 = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
1435 |
1436 if iter.flag then
1437     let e:String = elem.event in
1438     if iter.i2 = effectSize and e = lastEffect then

```



```

1492 effectSize:Integer = effects->size(),
1493 firstEffect:String = effects->first(),
1494 lastEffect:String = effects->last(),
1495 secondEffectDistance:Integer = effectDistances->at(2).value
1496 in
1497 subtrace->iterate(elem:trace::TraceElement;
1498   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
1499 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:
      Integer = 0}
1500 |
1501 if iter.flag then
1502   let e:String = elem.event in
1503   if iter.midCriticalInstant > 0 and elem.timestamp >= iter.midCriticalInstant then
1504     Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, i2:Integer = null,
          effectCriticalInstant:Integer = null} // satisfaction
1505   else
1506     if iter.i2 = effectSize and e = lastEffect then
1507       Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, i2:Integer = null,
          effectCriticalInstant:Integer = null} // violation
1508     else
1509       if iter.i1 > 1 and e = causes->at(iter.i1) then
1510         if iter.i1 = causeSize then
1511           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer =
              1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1512         else
1513           if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
              effectDistances->at(iter.i2).which) then
1514             let i22:Integer = iter.i2 + 1 in
1515             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}
1516           else
1517             if e = firstEffect then
1518               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
1519             else
1520               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
                  iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1521             endif
1522           endif
1523         endif
1524       else
1525         if iter.midCriticalInstant = 0 and e = firstCause then
1526           if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
              effectDistances->at(iter.i2).which) then
1527             let i22:Integer = iter.i2 + 1 in
1528             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
1529             else
1530               if e = firstEffect then
1531                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                    i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a
                    potential violation
1532               else
1533                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                    i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1534               endif
1535             endif
1536           else
1537             if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
              effectDistances->at(iter.i2).which) then
1538               let i22:Integer = iter.i2 + 1 in
1539               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
1540             else
1541               if e = firstEffect then //midCriticalInstant is either 0 or midCriticalInstant > elem.timestamp
1542                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                    i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a

```

```

        potential violation
1543     else
1544         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
            i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1545     endif
1546     endif
1547     endif
1548     endif
1549     endif
1550     endif
1551     else
1552         iter
1553     endif
1554 ).midCriticalInstant >= -1
1555
1556 =====
1557 def: checkPatternPrecedenceManyManyAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):
    Boolean =
1558 let
1559     causeSize:Integer = causes->size(),
1560     firstCause:String = causes->first(),
1561     effectSize:Integer = effects->size(),
1562     firstEffect:String = effects->first(),
1563     lastEffect:String = effects->last(),
1564     secondEffectDistance:Integer = effectDistances->at(2).value
1565 in
1566 subtrace->iterate(elem:trace::TraceElement;
1567     iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
1568     = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:
        Integer = 0}
1569 |
1570 if iter.flag then
1571     let e:String = elem.event in
1572     if iter.i2 = effectSize and e = lastEffect then
1573         Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null,
            effectCriticalInstant:Integer = null}
1574     else
1575         if iter.i1 > 1 and e = causes->at(iter.i1) then
1576             if iter.i1 = causeSize then
1577                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
                    i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1578             else
1579                 if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
                    effectDistances->at(iter.i2).which) then
1580                     let i22:Integer = iter.i2 + 1 in
1581                     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}
1582                 else
1583                     if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1584                         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter
                            .i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
1585                     else
1586                         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter
                            .i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1587                     endif
1588                 endif
1589             endif
1590         else
1591             if e = firstCause then
1592                 if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
                    effectDistances->at(iter.i2).which) then
1593                     let i22:Integer = iter.i2 + 1 in
1594                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
                        Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
1595                 else
1596                     if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1597                         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,

```

```

        i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
1598     else
1599         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
        i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1600     endif
1601 endif
1602 else
1603     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
1604         let i22:Integer = iter.i2 + 1 in
1605         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
        Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
1606     else
1607         if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1608             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
        i2:Integer = 2, effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance}
1609         else
1610             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
        i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1611         endif
1612     endif
1613 endif
1614 endif
1615 endif
1616 else
1617     iter
1618 endif
1619 ).flag
1620
1621 =====
1622 def: checkPatternPrecedenceManyManyExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
        midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):
        Boolean =
1623 let
1624     causeSize:Integer = causes->size(),
1625     firstCause:String = causes->first(),
1626     effectSize:Integer = effects->size(),
1627     firstEffect:String = effects->first(),
1628     lastEffect:String = effects->last(),
1629     secondEffectDistance:Integer = effectDistances->at(2).value
1630 in
1631 subtrace->iterate(elem:trace::TraceElement;
1632     iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, i2:Integer, effectCriticalInstant:
        Integer)
1633     = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1, i2:Integer = 1,
        effectCriticalInstant:Integer = 0}
1634 |
1635 if iter.flag then
1636     let e:String = elem.event in
1637     if iter.i2 = effectSize and e = lastEffect then
1638         Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null, i2:Integer = null,
        effectCriticalInstant:Integer = null}
1639     else
1640         if iter.i1 > 1 and e = causes->at(iter.i1) then
1641             if iter.i1 = causeSize then
1642                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(
        elem.timestamp+midDistance), i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.
        effectCriticalInstant}
1643             else
1644                 if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
1645                     let i22:Integer = iter.i2 + 1 in
1646                     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}
1647                 else
1648                     if e = firstEffect then
1649                         let t:Integer = elem.timestamp in
1650                         if iter.midCriticalInstants->includes(t) then

```

```

1651         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}
1652     else
1653         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
            Integer = iter.i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
1654     endif
1655     else
1656         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
            Integer = iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1657     endif
1658     endif
1659     endif
1660     else
1661         if e = firstCause then
1662             if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
                effectDistances->at(iter.i2).which) then
1663                 let i22:Integer = iter.i2 + 1 in
1664                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                    Integer = 2, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(
                    i22).value}
1665             else
1666                 if e = firstEffect then
1667                     let t:Integer = elem.timestamp in
1668                     if iter.midCriticalInstants->includes(t) then
1669                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
                            select(subElem | subElem > t), i1:Integer = 2, i2:Integer = 1, effectCriticalInstant:Integer =
                            iter.effectCriticalInstant}
1670                     else
1671                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                            Integer = 2, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
1672                     endif
1673                     else
1674                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                            Integer = 2, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1675                     endif
1676                 endif
1677             else
1678                 if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
                    effectDistances->at(iter.i2).which) then
1679                     let i22:Integer = iter.i2 + 1 in
1680                     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                        Integer = 1, i2:Integer = i22, effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(
                        i22).value}
1681                 else
1682                     if e = firstEffect then
1683                         let t:Integer = elem.timestamp in
1684                         if iter.midCriticalInstants->includes(t) then
1685                             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
                                    select(subElem | subElem > t), i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer =
                                    iter.effectCriticalInstant}
1686                             else
1687                                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                                    Integer = 1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
1688                             endif
1689                             else
1690                                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                                    Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
1691                             endif
1692                         endif
1693                     endif
1694                 endif
1695             endif
1696         else
1697             iter
1698         endif
1699     ).flag
1700
1701     =====

```

```

1702 def: checkPatternPrecedenceManyManyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      midDistance:TemPsy::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:
      Integer))):Boolean =
1703 let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
1704 if midWhich = TemPsy::ComparingOperator::ATLEAST then
1705   self.checkPatternPrecedenceManyManyAtLeastMidRight(subtrace, causes, midValue, effects, effectDistances)
1706 else
1707   if midWhich = TemPsy::ComparingOperator::ATMOST then
1708     self.checkPatternPrecedenceManyManyAtMostMidRight(subtrace, causes, midValue, effects, effectDistances)
1709   else
1710     self.checkPatternPrecedenceManyManyExactlyMidRight(subtrace, causes, midValue, effects, effectDistances)
1711   endif
1712 endif
1713
1714 =====
1715 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 =
1716 let
1717   causeSize:Integer = causes->size(),
1718   firstCause:String = causes->first(),
1719   secondCauseDistance:Integer = causeDistances->at(2).value,
1720   effectSize:Integer = effects->size(),
1721   firstEffect:String = effects->first(),
1722   lastEffect:String = effects->last(),
1723   secondEffectDistance:Integer = effectDistances->at(2).value
1724 in
1725 subtrace->iterate(elem:trace::TraceElement;
1726   iter:Tuple(flag:Boolean, result:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer,
      effectCriticalInstant:Integer)
1727   = Tuple{flag:Boolean = true, result:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1,
      effectCriticalInstant:Integer = 0}
1728   |
1729   if iter.flag then
1730     let e:String = elem.event in
1731     if iter.i2 = effectSize and e = lastEffect then
1732       Tuple{flag:Boolean = false, result:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
         Integer = null, effectCriticalInstant:Integer = null}
1733     else
1734       if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
         causeDistances->at(iter.i1).which) then
1735         if iter.i1 = causeSize then
1736           Tuple{flag:Boolean = false, result:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer = null, i2
             :Integer = null, effectCriticalInstant:Integer = null}
1737         else
1738           let i11:Integer = iter.i1 + 1 in
1739           if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
             effectDistances->at(iter.i2).which) then
1740             let i22:Integer = iter.i2 + 1 in
1741             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}
1742           else
1743             if e = firstEffect then
1744               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}
1745             else
1746               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}
1747             endif
1748           endif
1749         endif
1750       else
1751         if e = firstCause then
1752           if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
             effectDistances->at(iter.i2).which) then
1753             let i22:Integer = iter.i2 + 1 in

```

```

1754     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}
1755     else
1756     if e = firstEffect then
1757     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}
1758     else
1759     Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 2, causeCriticalInstant:
        Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1, effectCriticalInstant:Integer =
        iter.effectCriticalInstant}
1760     endif
1761     endif
1762     else
1763     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
1764     let i22:Integer = iter.i2 + 1 in
1765     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}
1766     else
1767     if e = firstEffect then
1768     Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:
        Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer = elem.
        timestamp + secondEffectDistance}
1769     else
1770     Tuple{flag:Boolean = iter.flag, result:Integer = iter.result, i1:Integer = 1, causeCriticalInstant:
        Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.
        effectCriticalInstant}
1771     endif
1772     endif
1773     endif
1774     endif
1775     endif
1776     else
1777     iter
1778     endif
1779 ).result >= -1
1780
1781 =====
1782 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 =
1783 let
1784 causeSize:Integer = causes->size(),
1785 firstCause:String = causes->first(),
1786 secondCauseDistance:Integer = causeDistances->at(2).value,
1787 effectSize:Integer = effects->size(),
1788 firstEffect:String = effects->first(),
1789 lastEffect:String = effects->last(),
1790 secondEffectDistance:Integer = effectDistances->at(2).value
1791 in
1792 subtrace->iterate(elem:trace::TraceElement;
1793 iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer,
        effectCriticalInstant:Integer)
1794 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
        Integer = 1, effectCriticalInstant:Integer = 0}
1795 |
1796 if iter.flag then
1797 let e:String = elem.event in
1798 if iter.midCriticalInstant > 0 and elem.timestamp >= iter.midCriticalInstant then
1799 Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:Integer =
        null, i2:Integer = null, effectCriticalInstant:Integer = null} // satisfaction
1800 else
1801 if iter.i2 = effectSize and e = lastEffect then
1802 Tuple{flag:Boolean = false, midCriticalInstant:Integer = -2, i1:Integer = null, causeCriticalInstant:Integer
        = null, i2:Integer = null, effectCriticalInstant:Integer = null} // violation
1803 else

```

```

1804   if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
1805       causeDistances->at(iter.i1).which) then
1806       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer =
1807         1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:
1808         Integer = iter.effectCriticalInstant}
1809     else
1810       let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11
1811         ).value in
1812       if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
1813         effectDistances->at(iter.i2).which) then
1814         let i22:Integer = iter.i2 + 1 in
1815         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
1816           causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = i22, effectCriticalInstant:
1817             Integer = elem.timestamp + effectDistances->at(i22).value}
1818       else
1819         if e = firstEffect then
1820           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
1821             i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2,
1822             effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a potential violation
1823         else
1824           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer =
1825             i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1,
1826             effectCriticalInstant:Integer = iter.effectCriticalInstant}
1827         endif
1828       endif
1829     endif
1830   else
1831     if iter.midCriticalInstant = 0 and e = firstCause then
1832       if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
1833         effectDistances->at(iter.i2).which) then
1834         let i22:Integer = iter.i2 + 1 in
1835         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
1836           causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = i22,
1837           effectCriticalInstant:Integer = elem.timestamp + effectDistances->at(i22).value}
1838       else
1839         if e = firstEffect then
1840           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
1841             causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2,
1842             effectCriticalInstant:Integer = elem.timestamp + secondEffectDistance} // a potential violation
1843         else
1844           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
1845             causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1,
1846             effectCriticalInstant:Integer = iter.effectCriticalInstant}
1847         endif
1848       endif
1849     endif
1850   else
1851     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
1852       effectDistances->at(iter.i2).which) then
1853       let i22:Integer = iter.i2 + 1 in
1854       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
1855         causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:
1856         Integer = elem.timestamp + effectDistances->at(i22).value}
1857     else
1858       if e = firstEffect then
1859         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
1860           causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant
1861           :Integer = elem.timestamp + secondEffectDistance} // a potential violation
1862       else
1863         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
1864           causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant
1865           :Integer = iter.effectCriticalInstant}
1866       endif
1867     endif
1868   endif
1869 endif
1870 endif
1871 endif
1872 endif
1873 else

```

```

1848   iter
1849   endif
1850 ).midCriticalInstant >= -1
1851
1852 =====
1853 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 =
1854 let
1855   causeSize:Integer = causes->size(),
1856   firstCause:String = causes->first(),
1857   secondCauseDistance:Integer = causeDistances->at(2).value,
1858   effectSize:Integer = effects->size(),
1859   firstEffect:String = effects->first(),
1860   lastEffect:String = effects->last(),
1861   secondEffectDistance:Integer = effectDistances->at(2).value
1862 in
1863 subtrace->iterate(elem:trace::TraceElement;
1864   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer,
      effectCriticalInstant:Integer)
1865   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
      Integer = 1, effectCriticalInstant:Integer = 0}
1866   |
1867   if iter.flag then
1868     let e:String = elem.event in
1869     if iter.i2 = effectSize and e = lastEffect then
1870       Tuple{flag:Boolean = false, midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer
          = null, i2:Integer = null, effectCriticalInstant:Integer = null}
1871     else
1872       if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
          causeDistances->at(iter.i1).which) then
1873         if iter.i1 = causeSize then
1874           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
              causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:
              Integer = iter.effectCriticalInstant}
1875         else
1876           let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).
              value in
1877           if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
              effectDistances->at(iter.i2).which) then
1878             let i22:Integer = iter.i2 + 1 in
1879             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}
1880           else
1881             if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1882               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
                  causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2, effectCriticalInstant:
                  Integer = elem.timestamp + secondEffectDistance}
1883             else
1884               Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
                  causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1, effectCriticalInstant:
                  Integer = iter.effectCriticalInstant}
1885             endif
1886           endif
1887         endif
1888       else
1889         if e = firstCause then
1890           if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
              effectDistances->at(iter.i2).which) then
1891             let i22:Integer = iter.i2 + 1 in
1892             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}
1893           else
1894             if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1895               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}

```



```

1896     else
1897         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
            causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1,
            effectCriticalInstant:Integer = iter.effectCriticalInstant}
1898     endif
1899     endif
1900     else
1901         if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
            effectDistances->at(iter.i2).which) then
1902             let i22:Integer = iter.i2 + 1 in
1903             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}
1904         else
1905             if e = firstEffect and elem.timestamp > iter.midCriticalInstant then
1906                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                    causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:
                    Integer = elem.timestamp + secondEffectDistance}
1907             else
1908                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                    causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:
                    Integer = iter.effectCriticalInstant}
1909             endif
1910         endif
1911     endif
1912     endif
1913     endif
1914     else
1915         iter
1916     endif
1917 ).flag
1918
1919 =====
1920 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 =
1921 let
1922     causeSize:Integer = causes->size(),
1923     firstCause:String = causes->first(),
1924     secondCauseDistance:Integer = causeDistances->at(2).value,
1925     effectSize:Integer = effects->size(),
1926     firstEffect:String = effects->first(),
1927     lastEffect:String = effects->last(),
1928     secondEffectDistance:Integer = effectDistances->at(2).value
1929 in
1930 subtrace->iterate(elem:trace::TraceElement;
1931     iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), i1:Integer, causeCriticalInstant:Integer, i2:
            Integer, effectCriticalInstant:Integer)
1932     = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, i1:Integer = 1,
            causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
1933     |
1934     if iter.flag then
1935         let e:String = elem.event in
1936         if iter.i2 = effectSize and e = lastEffect then
1937             Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = null, i1:Integer = null,
                causeCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
1938         else
1939             if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
                causeDistances->at(iter.i1).which) then
1940                 if iter.i1 = causeSize then
1941                     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}
1942                 else
1943                     let i11:Integer = iter.i1 + 1, nextCauseCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).
                        value in
1944                     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
                        effectDistances->at(iter.i2).which) then
1945                         let i22:Integer = iter.i2 + 1 in

```

```

1946     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}
1947     else
1948     if e = firstEffect then
1949     let t:Integer = elem.timestamp in
1950     if iter.midCriticalInstants->includes(t) then
1951     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}
1952     else
1953     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
          Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 2,
          effectCriticalInstant:Integer = t + secondEffectDistance}
1954     endif
1955     else
1956     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
          Integer = i11, causeCriticalInstant:Integer = nextCauseCriticalInstant, i2:Integer = 1,
          effectCriticalInstant:Integer = iter.effectCriticalInstant}
1957     endif
1958     endif
1959     endif
1960     else
1961     if e = firstCause then
1962     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
          effectDistances->at(iter.i2).which) then
1963     let i22:Integer = iter.i2 + 1 in
1964     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}
1965     else
1966     if e = firstEffect then
1967     let t:Integer = elem.timestamp in
1968     if iter.midCriticalInstants->includes(t) then
1969     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}
1970     else
1971     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}
1972     endif
1973     else
1974     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}
1975     endif
1976     endif
1977     else
1978     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
          effectDistances->at(iter.i2).which) then
1979     let i22:Integer = iter.i2 + 1 in
1980     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}
1981     else
1982     if e = firstEffect then
1983     let t:Integer = elem.timestamp in
1984     if iter.midCriticalInstants->includes(t) then
1985     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
          }
1986     else
1987     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
          Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2,
          effectCriticalInstant:Integer = t + secondEffectDistance}
1988     endif

```

```

1989         else
1990             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants, i1:
                Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1,
                effectCriticalInstant:Integer = iter.effectCriticalInstant}
1991         endif
1992     endif
1993 endif
1994 endif
1995 endif
1996 else
1997     iter
1998 endif
1999 ).flag
2000
2001 =====
2002 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 =
2003 let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator=midDistance.comparingOperator in
2004 if midWhich = TemPsy::ComparingOperator::ATLEAST then
2005     self.checkPatternPrecedenceManyManyLeftAtLeastMidRight(subtrace, causes, causeDistances, midValue, effects,
        effectDistances)
2006 else
2007     if midWhich = TemPsy::ComparingOperator::ATMOST then
2008         self.checkPatternPrecedenceManyManyLeftAtMostMidRight(subtrace, causes, causeDistances, midValue, effects,
            effectDistances)
2009     else
2010         self.checkPatternPrecedenceManyManyLeftExactlyMidRight(subtrace, causes, causeDistances, midValue, effects,
            effectDistances)
2011     endif
2012 endif
2013
2014 =====
2015 def: checkPatternResponse(subtrace:OrderedSet(trace::TraceElement), pattern:TemPsy::Pattern):Boolean =
2016 --check the satisfiability of the response pattern 'effect responding cause'
2017 --in the first event in the chain 'effect', it may contains time distance to the last event in the chain 'cause'
2018 if subtrace->isEmpty() then
2019     true
2020 else
2021     let orderPattern:TemPsy::OrderPattern = pattern.oclAsType(TemPsy::OrderPattern),
2022         causes:Sequence(String) = orderPattern.block2.event.name,
2023         causeDistances:Sequence(Tuple(which:Integer, value:Integer)) = self.loadDistances(orderPattern.block2.
            timeDistance),
2024         causeSize:Integer = causes->size(),
2025         effects:Sequence(String) = orderPattern.block1.event.name,
2026         effectDistances:Sequence(Tuple(which:Integer, value:Integer)) = self.loadDistances(orderPattern.block1.
            timeDistance),
2027         effectSize:Integer = effects->size()
2028     in
2029     if causeDistances->isEmpty() then
2030         if effectDistances->isEmpty() then
2031             if orderPattern.timeDistance->isEmpty() then
2032                 if causeSize = 1 then
2033                     let cause:String = causes->first() in
2034                     if effectSize = 1 then
2035                         let effect:String = effects->first() in
2036                         self.checkPatternResponseOneOnePlain(subtrace, cause, effect)
2037                     else
2038                         self.checkPatternResponseOneManyPlain(subtrace, cause, effects)
2039                     endif
2040                 else
2041                     if effectSize = 1 then
2042                         let effect:String = effects->first() in
2043                         self.checkPatternResponseManyOnePlain(subtrace, causes, effect)
2044                     else
2045                         self.checkPatternResponseManyManyPlain(subtrace, causes, effects)
2046                     endif
2047                 endif
2048             else

```

```

2049     if causeSize = 1 then
2050     let cause:String = causes->first() in
2051     if effectSize = 1 then
2052         let effect:String = effects->first() in
2053         self.checkPatternResponseOneOneMid(subtrace, cause, orderPattern.timeDistance, effect)
2054     else
2055         self.checkPatternResponseOneManyMid(subtrace, cause, orderPattern.timeDistance, effects)
2056     endif
2057 else
2058     if effectSize = 1 then
2059         let effect:String = effects->first() in
2060         self.checkPatternResponseManyOneMid(subtrace, causes, orderPattern.timeDistance, effect)
2061     else
2062         self.checkPatternResponseManyManyMid(subtrace, causes, orderPattern.timeDistance, effects)
2063     endif
2064 endif
2065 endif
2066 else
2067     if orderPattern.timeDistance->isEmpty() then
2068         if causeSize = 1 then
2069             let cause:String = causes->first() in
2070             self.checkPatternResponseOneManyRight(subtrace, cause, effects, effectDistances)
2071         else
2072             self.checkPatternResponseManyManyRight(subtrace, causes, effects, effectDistances)
2073         endif
2074     else
2075         if causeSize = 1 then
2076             let cause:String = causes->first() in
2077             self.checkPatternResponseOneManyMidRight(subtrace, cause, orderPattern.timeDistance, effects,
                effectDistances)
2078         else
2079             self.checkPatternResponseManyManyMidRight(subtrace, causes, orderPattern.timeDistance, effects,
                effectDistances)
2080         endif
2081     endif
2082 endif
2083 else
2084     if effectDistances->isEmpty() then
2085         if orderPattern.timeDistance->isEmpty() then
2086             if effectSize = 1 then
2087                 let effect:String = effects->first() in
2088                 self.checkPatternResponseManyOneLeft(subtrace, causes, causeDistances, effect)
2089             else
2090                 self.checkPatternResponseManyManyLeft(subtrace, causes, causeDistances, effects)
2091             endif
2092         else
2093             if effectSize = 1 then
2094                 let effect:String = effects->first() in
2095                 self.checkPatternResponseManyOneLeftMid(subtrace, causes, causeDistances, orderPattern.timeDistance, effect
                )
2096             else
2097                 self.checkPatternResponseManyManyLeftMid(subtrace, causes, causeDistances, orderPattern.timeDistance,
                effects)
2098             endif
2099         endif
2100     else
2101         if orderPattern.timeDistance->isEmpty() then
2102             self.checkPatternResponseManyManyLeftRight(subtrace, causes, causeDistances, effects, effectDistances)
2103         else
2104             self.checkPatternResponseManyManyLeftMidRight(subtrace, causes, causeDistances, orderPattern.timeDistance,
                effects, effectDistances)
2105         endif
2106     endif
2107 endif
2108 endif
2109
2110 =====
2111 def: checkPatternResponseOneOnePlain(subtrace:OrderedSet(trace::TraceElement), cause:String, effect:String):Boolean =
2112 subtrace->iterate(

```

```

2113 elem:trace::TraceElement;
2114 result:Boolean = true
2115 |
2116 let e:String = elem.event in
2117 if e = cause then
2118   false
2119 else
2120   if e = effect then
2121     true
2122   else
2123     result
2124   endif
2125 endif
2126 )
2127
2128 =====
2129 def: checkPatternResponseOneOneAtLeastMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effect:String):Boolean =
2130 subtrace->iterate(elem:trace::TraceElement;
2131   midCriticalInstant:Integer = 0
2132   |
2133   let e:String = elem.event in
2134   if e = cause then
2135     elem.timestamp + distance
2136   else
2137     if e = effect and elem.timestamp >= midCriticalInstant then
2138       0
2139     else
2140       midCriticalInstant
2141     endif
2142   endif
2143 ) = 0
2144
2145 =====
2146 def: checkPatternResponseOneOneAtMostMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effect:String):Boolean =
2147 subtrace->iterate(elem:trace::TraceElement;
2148   iter:Tuple(flag:Boolean, midCriticalInstant:Integer)
2149   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0}
2150   |
2151   if iter.flag then
2152     let e:String = elem.event in
2153     if iter.midCriticalInstant = 0 and e = cause then
2154       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance}
2155     else
2156       if e = effect then
2157         if elem.timestamp <= iter.midCriticalInstant then
2158           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0}
2159         else
2160           Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1} // violation
2161         endif
2162       else
2163         iter
2164       endif
2165     endif
2166   else
2167     iter
2168   endif
2169 ).midCriticalInstant = 0
2170
2171 =====
2172 def: checkPatternResponseOneOneExactlyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effect:String):Boolean =
2173 subtrace->iterate(elem:trace::TraceElement;
2174   iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer))
2175   = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}}
2176   |
2177   if iter.flag then
2178     let e:String = elem.event in

```

```

2179 if e = cause then
2180   Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(elem.
      timestamp + distance)}
2181 else
2182   if e = effect and iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstants->first()
      then
2183     let t:Integer = elem.timestamp in
2184     if t = iter.midCriticalInstants->first() then
2185       Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding
          (t)}
2186     else
2187       Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants}
2188     endif
2189   else
2190     iter
2191   endif
2192 endif
2193 else
2194   iter
2195 endif
2196 ).midCriticalInstants->isEmpty()
2197
2198 def: checkPatternResponseOneOneMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:TempPsy::
      TimeDistance, effect:String):Boolean =
2199 let value:Integer = distance.value, which:TempPsy::ComparingOperator = distance.comparingOperator in
2200 if which = TempPsy::ComparingOperator::ATLEAST then
2201   self.checkPatternResponseOneOneAtLeastMid(subtrace, cause, value, effect)
2202 else
2203   if which = TempPsy::ComparingOperator::ATMOST then
2204     self.checkPatternResponseOneOneAtMostMid(subtrace, cause, value, effect)
2205   else
2206     self.checkPatternResponseOneOneExactlyMid(subtrace, cause, value, effect)
2207   endif
2208 endif
2209
2210 =====
2211 def: checkPatternResponseOneManyPlain(subtrace:OrderedSet(trace::TraceElement), cause:String, effects:Sequence(String
      )):Boolean =
2212 let
2213   effectSize:Integer = effects->size(),
2214   firstEffect:String = effects->first()
2215 in
2216 subtrace->iterate(
2217   elem:trace::TraceElement;
2218   iter:Tuple(flag:Boolean, i2:Integer) = Tuple(flag:Boolean = true, i2:Integer = 1)
2219   |
2220   let e:String = elem.event in
2221   if e = cause then
2222     Tuple(flag:Boolean = false, i2:Integer = 1)
2223   else
2224     if not iter.flag then
2225       if e = effects->at(iter.i2) then
2226         if iter.i2 = effectSize then
2227           Tuple(flag:Boolean = true, i2:Integer = 1)
2228         else
2229           Tuple(flag:Boolean = iter.flag, i2:Integer = iter.i2 + 1)
2230         endif
2231       else
2232         if e = firstEffect then
2233           Tuple(flag:Boolean = iter.flag, i2:Integer = 2)
2234         else
2235           Tuple(flag:Boolean = iter.flag, i2:Integer = 1)
2236         endif
2237       endif
2238     else
2239       iter
2240     endif
2241   endif
2242 ).flag

```

```

2243
2244 =====
2245 def: checkPatternResponseOneManyAtLeastMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effects:Sequence(String)):Boolean =
2246 let
2247   effectSize:Integer = effects->size(),
2248   firstEffect:String = effects->first()
2249 in
2250 subtrace->iterate(elem:trace::TraceElement;
2251   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer)
2252   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1}
2253   |
2254   let e:String = elem.event in
2255   if e = cause then // latest cause
2256     Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + distance, i2:Integer = 1}
2257   else
2258     if not iter.flag then
2259       if iter.i2 > 1 and e = effects->at(iter.i2) then
2260         if iter.i2 = effectSize then // until effects->last(), the property is satisfied so far
2261           Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
2262         else
2263           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2
2264             + 1}
2265         endif
2266       else
2267         if e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
2268           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
2269         else
2270           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
2271         endif
2272       else
2273         iter
2274       endif
2275     endif
2276   ).flag
2277
2278 =====
2279 def: checkPatternResponseOneManyAtMostMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effects:Sequence(String)):Boolean =
2280 let
2281   effectSize:Integer = effects->size(),
2282   firstEffect:String = effects->first()
2283 in
2284 subtrace->iterate(elem:trace::TraceElement;
2285   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer)
2286   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1}
2287   |
2288   let e:String = elem.event in
2289   if iter.flag then
2290     if iter.midCriticalInstant = 0 then
2291       if e = cause then
2292         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i2:Integer = 1}
2293       else
2294         iter
2295       endif
2296     else
2297       if iter.i2 > 1 and e = effects->at(iter.i2) then
2298         if iter.i2 = effectSize then
2299           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i2:Integer = 1}
2300         else
2301           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2
2302             + 1}
2303         endif
2304       else
2305         if e = firstEffect then
2306           if elem.timestamp <= iter.midCriticalInstant then
2307             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
2308           else

```

```

2308     Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i2:Integer = null}
2309     endif
2310     else
2311         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1}
2312     endif
2313     endif
2314     endif
2315     else
2316         iter
2317     endif
2318 ).midCriticalInstant = 0
2319
2320 =====
2321 def: checkPatternResponseOneManyExactlyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:Integer,
      effects:Sequence(String)):Boolean =
2322 let
2323     effectSize:Integer = effects->size(),
2324     firstEffect:String = effects->first()
2325 in
2326 subtrace->iterate(elem:trace::TraceElement;
2327     iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i2:Integer)
2328     = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i2
      :Integer = 1}
2329 |
2330 if iter.flag then
2331     let e:String = elem.event in
2332     if e = cause then
2333         let ct:Integer = elem.timestamp + distance in
2334         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct),
          midCriticalInstant:Integer = ct, i2:Integer = 1}
2335     else
2336         if iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstant then
2337             if iter.i2 > 1 and e = effects->at(iter.i2) then
2338                 if iter.i2 = effectSize then
2339                     if iter.midCriticalInstants->size() = 1 then
2340                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
          excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i2:
          Integer = 1}
2341                     else
2342                         let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
2343                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
          excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i2:Integer =
          1}
2344                     endif
2345                 else
2346                     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
          midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = iter.i2 + 1}
2347                 endif
2348             else
2349                 if e = firstEffect and elem.timestamp = iter.midCriticalInstant then
2350                     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
          midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2}
2351                 else
2352                     Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
          midCriticalInstant:Integer = null, i2:Integer = null}
2353                 endif
2354             endif
2355         else
2356             iter
2357         endif
2358     endif
2359     else
2360         iter
2361     endif
2362 ).midCriticalInstants->isEmpty()
2363
2364 =====
2365 def: checkPatternResponseOneManyMid(subtrace:OrderedSet(trace::TraceElement), cause:String, distance:TempPsy::
      TimeDistance, effects:Sequence(String)):Boolean =

```



```

2366 let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
2367 if which = TemPsy::ComparingOperator::ATLEAST then
2368   self.checkPatternResponseOneManyAtLeastMid(subtrace, cause, value, effects)
2369 else
2370   if which = TemPsy::ComparingOperator::ATMOST then
2371     self.checkPatternResponseOneManyAtMostMid(subtrace, cause, value, effects)
2372   else
2373     self.checkPatternResponseOneManyExactlyMid(subtrace, cause, value, effects)
2374   endif
2375 endif
2376
2377 =====
2378 def: checkPatternResponseManyOnePlain(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effect:
      String):Boolean =
2379 let
2380   causeSize:Integer = causes->size(),
2381   firstCause:String = causes->first()
2382 in
2383 subtrace->iterate(elem:trace::TraceElement;
2384   iter:Tuple(flag:Boolean, i1:Integer) = Tuple{flag:Boolean = true, i1:Integer = 1}
2385   |
2386   let e:String = elem.event in
2387   if iter.i1 > 1 and e = causes->at(iter.i1) then
2388     if iter.i1 = causeSize then
2389       Tuple{flag:Boolean = false, i1:Integer = 1}
2390     else
2391       Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1}
2392     endif
2393   else
2394     if e = firstCause then
2395       Tuple{flag:Boolean = iter.flag, i1:Integer = 2}
2396     else
2397       if e = effect then
2398         Tuple{flag:Boolean = true, i1:Integer = 1}
2399       else
2400         Tuple{flag:Boolean = iter.flag, i1:Integer = 1}
2401       endif
2402     endif
2403   endif
2404 ).flag
2405
2406 =====
2407 def: checkPatternResponseManyOneAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      distance:Integer, effect:String):Boolean =
2408 let
2409   causeSize:Integer = causes->size(),
2410   firstCause:String = causes->first()
2411 in
2412 subtrace->iterate(elem:trace::TraceElement;
2413   iter:Tuple(midCriticalInstant:Integer, i1:Integer) = Tuple{midCriticalInstant:Integer = 0, i1:Integer = 1}
2414   |
2415   let e:String = elem.event in
2416   if iter.i1 > 1 and e = causes->at(iter.i1) then
2417     if iter.i1 = causeSize then
2418       Tuple{midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1}
2419     else
2420       Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1}
2421     endif
2422   else
2423     if e = firstCause then
2424       Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
2425     else
2426       if e = effect and elem.timestamp >= iter.midCriticalInstant then
2427         Tuple{midCriticalInstant:Integer = 0, i1:Integer = 1}
2428       else
2429         Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
2430       endif
2431     endif
2432   endif

```

```

2433 ).midCriticalInstant = 0
2434
2435 =====
2436 def: checkPatternResponseManyOneAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance
      :Integer, effect:String):Boolean =
2437 let
2438   causeSize:Integer = causes->size(),
2439   firstCause:String = causes->first()
2440 in
2441 subtrace->iterate(elem:trace::TraceElement;
2442   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer)
2443   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1} |
2444   let e:String = elem.event in
2445   if iter.flag then
2446     if iter.midCriticalInstant = 0 then
2447       if iter.i1 > 1 and e = causes->at(iter.i1) then
2448         if iter.i1 = causeSize then
2449           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1}
2450         else
2451           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1
2452             + 1}
2453         endif
2454       else
2455         if e = firstCause then
2456           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}
2457         else
2458           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
2459         endif
2460       endif
2461     else
2462       if e = effect then
2463         if elem.timestamp <= iter.midCriticalInstant then
2464           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = 1}
2465         else
2466           Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null}
2467         endif
2468       else
2469         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
2470       endif
2471     endif
2472   iter
2473   endif
2474 ).midCriticalInstant = 0
2475
2476 =====
2477 def: checkPatternResponseManyOneExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      distance:Integer, effect:String):Boolean =
2478 let causeSize:Integer = causes->size(), firstCause:String = causes->first() in
2479 subtrace->iterate(elem:trace::TraceElement;
2480   iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer)
2481   = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1
2482     :Integer = 1}
2483   |
2484   if iter.flag then
2485     let e:String = elem.event, t:Integer = elem.timestamp in
2486     if iter.i1 > 1 and e = causes->at(iter.i1) then
2487       if iter.i1 = causeSize then
2488         let ct:Integer = t + distance in
2489         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct),
2490           midCriticalInstant:Integer = ct, i1:Integer = 1}
2491       else
2492         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
2493           midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1}
2494       endif
2495     else
2496       if e = firstCause then
2497         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
2498           midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2}

```

```

2495     else
2496     if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
2497     if t = iter.midCriticalInstant and e = effect then
2498     if iter.midCriticalInstants->size() = 1 then
2499     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
        excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i1:
        Integer = 1}
2500     else
2501     let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
2502     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
        excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i1:Integer =
        1}
2503     endif
2504     else
2505     Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
        midCriticalInstant:Integer = null, i1:Integer = null}
2506     endif
2507     else
2508     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
        midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1}
2509     endif
2510     endif
2511     endif
2512     else
2513     iter
2514     endif
2515 ) .midCriticalInstants->isEmpty()
2516
2517 def: checkPatternResponseManyOneMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:
    TempPsy::TimeDistance, effect:String):Boolean =
2518 let value:Integer = distance.value, which:TempPsy::ComparingOperator = distance.comparingOperator in
2519 if which = TempPsy::ComparingOperator::ATLEAST then
2520 self.checkPatternResponseManyOneAtLeastMid(subtrace, causes, value, effect)
2521 else
2522 if which = TempPsy::ComparingOperator::ATMOST then
2523 self.checkPatternResponseManyOneAtMostMid(subtrace, causes, value, effect)
2524 else
2525 self.checkPatternResponseManyOneExactlyMid(subtrace, causes, value, effect)
2526 endif
2527 endif
2528
2529 =====
2530 def: checkPatternResponseManyManyPlain(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effects:
    Sequence(String)):Boolean =
2531 let
2532 causeSize:Integer = causes->size(),
2533 firstCause:String = causes->first(),
2534 effectSize:Integer = effects->size(),
2535 firstEffect:String = effects->first(),
2536 lastEffect:String = effects->last()
2537 in
2538 subtrace->iterate(elem:trace::TraceElement;
2539 iter:Tuple(flag:Boolean, i1:Integer, i2:Integer) = Tuple{flag:Boolean = true, i1:Integer = 1, i2:Integer = 1}
2540 |
2541 let e:String = elem.event in
2542 if iter.i2 = effectSize and e = lastEffect then
2543 Tuple{flag:Boolean = true, i1:Integer = 1, i2:Integer = 1}
2544 else
2545 if iter.i1 > 1 and e = causes->at(iter.i1) then
2546 if iter.i1 = causeSize then
2547 Tuple{flag:Boolean = false, i1:Integer = 1, i2:Integer = 1}
2548 else
2549 if iter.i2 > 1 and e = effects->at(iter.i2) then
2550 Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = iter.i2 + 1}
2551 else
2552 if not iter.flag and e = firstEffect then
2553 Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = 2}
2554 else
2555 Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = 1}

```

```

2556     endif
2557   endif
2558 endif
2559 else
2560   if e = firstCause then
2561     if iter.i2 > 1 and e = effects->at(iter.i2) then
2562       Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = iter.i2 + 1}
2563     else
2564       if not iter.flag and e = firstEffect then
2565         Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = 2}
2566       else
2567         Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = 1}
2568       endif
2569     endif
2570   else
2571     if iter.i2 > 1 and e = effects->at(iter.i2) then
2572       Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = iter.i2 + 1}
2573     else
2574       if not iter.flag and e = firstEffect then
2575         Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = 2}
2576       else
2577         Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = 1}
2578       endif
2579     endif
2580   endif
2581 endif
2582 endif
2583 ).flag
2584
2585 =====
2586 def: checkPatternResponseManyManyAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
2587     distance:Integer, effects:Sequence(String)):Boolean =
2588 let
2589   causeSize:Integer = causes->size(),
2590   firstCause:String = causes->first(),
2591   effectSize:Integer = effects->size(),
2592   firstEffect:String = effects->first(),
2593   lastEffect:String = effects->last()
2594 in
2595 subtrace->iterate(elem:trace::TraceElement;
2596   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer)
2597 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1}
2598 |
2599 let e:String = elem.event in
2600 if iter.i2 = effectSize and e = lastEffect then
2601   Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1}
2602 else
2603   if iter.i1 > 1 and e = causes->at(iter.i1) then
2604     if iter.i1 = causeSize then
2605       Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1, i2:
2606         Integer = 1}
2607     else
2608       if iter.i2 > 1 and e = effects->at(iter.i2) then
2609         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1
2610           + 1, i2:Integer = iter.i2 + 1}
2611       else
2612         if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
2613           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
2614             i1 + 1, i2:Integer = 2}
2615         else
2616           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
2617             i1 + 1, i2:Integer = 1}
2618         endif
2619       endif
2620     endif
2621   endif
2622 else
2623   if e = firstCause then
2624     if iter.i2 > 1 and e = effects->at(iter.i2) then
2625       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:

```

```

        Integer = iter.i2 + 1}
2620 else
2621   if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
2622     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
        Integer = 2}
2623   else
2624     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
        Integer = 1}
2625   endif
2626 endif
2627 else
2628   if iter.i2 > 1 and e = effects->at(iter.i2) then
2629     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
        Integer = iter.i2 + 1}
2630   else
2631     if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
2632       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
        Integer = 2}
2633     else
2634       Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
        Integer = 1}
2635     endif
2636   endif
2637 endif
2638 endif
2639 endif
2640 ).flag
2641
2642 =====
2643 def: checkPatternResponseManyManyAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
        distance:Integer, effects:Sequence(String)):Boolean =
2644 let
2645   causeSize:Integer = causes->size(),
2646   firstCause:String = causes->first(),
2647   effectSize:Integer = effects->size(),
2648   firstEffect:String = effects->first()
2649 in
2650 subtrace->iterate(elem:trace::TraceElement;
2651   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer)
2652 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1}
2653 |
2654   let e:String = elem.event in
2655   if iter.flag then
2656     if iter.midCriticalInstant = 0 then
2657       if iter.i1 > 1 and e = causes->at(iter.i1) then
2658         if iter.i1 = causeSize then
2659           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + distance, i1:Integer = 1, i2:
            Integer = iter.i2}
2660         else
2661           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1
            + 1, i2:Integer = iter.i2}
2662         endif
2663       else
2664         if e = firstCause then
2665           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
            Integer = iter.i2}
2666         else
2667           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
            Integer = iter.i2}
2668         endif
2669       endif
2670     else
2671       if iter.i2 > 1 and e = effects->at(iter.i2) then
2672         if iter.i2 = effectSize then
2673           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = iter.i1, i2:Integer = 1}
2674         else
2675           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1,
            i2:Integer = iter.i2 + 1}
2676         endif

```

```

2677     else
2678         if e = firstEffect then
2679             if elem.timestamp <= iter.midCriticalInstant then
2680                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
                    i1, i2:Integer = 2}
2681             else
2682                 Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, i2:Integer = null}
2683             endif
2684         else
2685             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1,
                    i2:Integer = 1}
2686         endif
2687     endif
2688 endif
2689 else
2690     iter
2691 endif
2692 ).midCriticalInstant = 0
2693
2694 =====
2695 def: checkPatternResponseManyManyExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
        distance:Integer, effects:Sequence(String)):Boolean =
2696 let
2697     causeSize:Integer = causes->size(),
2698     firstCause:String = causes->first(),
2699     effectSize:Integer = effects->size(),
2700     firstEffect:String = effects->first(),
2701     lastEffect:String = effects->last()
2702 in
2703 subtrace->iterate(elem:trace::TraceElement;
2704     iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer, i2:Integer)
2705     = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1
        :Integer = 1, i2:Integer = 1} |
2706     if iter.flag then
2707         let e:String = elem.event in
2708         if iter.i2 = effectSize and e = lastEffect then
2709             if iter.midCriticalInstants->size() = 1 then
2710                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(
                    iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
                    Integer = 1}
2711             else
2712                 let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
2713                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->excluding(
                    iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i1:Integer = 1, i2:Integer =
                    1}
2714             endif
2715         else
2716             if iter.i1 > 1 and e = causes->at(iter.i1) then
2717                 if iter.i1 = causeSize then
2718                     let ct:Integer = elem.timestamp + distance in
2719                     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct
                        ), midCriticalInstant:Integer = ct, i1:Integer = 1, i2:Integer = 1}
2720                 else
2721                     if iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstant then
2722                         if iter.i2 > 1 and e = effects->at(iter.i2) then
2723                             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                                    midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = iter.
                                    i2 + 1}
2724                         else
2725                             if e = firstEffect and elem.timestamp = iter.midCriticalInstant then
2726                                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                                        midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 2}
2727                             else
2728                                 Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                                        midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null}
2729                             endif
2730                         endif
2731                     else
2732                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,

```

```

        midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1 + 1, i2:Integer = 1}
2733     endif
2734   endif
2735   else
2736     if e = firstCause then
2737       if iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstant then
2738         if iter.i2 > 1 and e = effects->at(iter.i2) then
2739           Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = iter.i2 + 1}
2740         else
2741           if e = firstEffect and elem.timestamp = iter.midCriticalInstant then
2742             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                    midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 2}
2743           else
2744             Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                    midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null}
2745           endif
2746         endif
2747       else
2748         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 1}
2749       endif
2750     else
2751       if iter.midCriticalInstants->notEmpty() and elem.timestamp >= iter.midCriticalInstant then
2752         if iter.i2 > 1 and e = effects->at(iter.i2) then
2753           Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = iter.i2 + 1}
2754         else
2755           if e = firstEffect and elem.timestamp = iter.midCriticalInstant then
2756             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                    midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2}
2757           else
2758             Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                    midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null}
2759           endif
2760         endif
2761       else
2762         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1}
2763       endif
2764     endif
2765   endif
2766   endif
2767   else
2768     iter
2769   endif
2770 ).midCriticalInstants->isEmpty()
2771
2772 =====
2773 def: checkPatternResponseManyManyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), distance:
      TemPsy::TimeDistance, effects:Sequence(String)):Boolean =
2774 let value:Integer = distance.value, which:TemPsy::ComparingOperator = distance.comparingOperator in
2775 if which = TemPsy::ComparingOperator::ATLEAST then
2776   self.checkPatternResponseManyManyAtLeastMid(subtrace, causes, value, effects)
2777 else
2778   if which = TemPsy::ComparingOperator::ATMOST then
2779     self.checkPatternResponseManyManyAtMostMid(subtrace, causes, value, effects)
2780   else
2781     self.checkPatternResponseManyManyExactlyMid(subtrace, causes, value, effects)
2782   endif
2783 endif
2784
2785 =====
2786 def: checkPatternResponseOneManyRight(subtrace:OrderedSet(trace::TraceElement), cause:String, effects:Sequence(String)
      ), effectDistances:Sequence(Tuple(which:Integer, value:Integer)):Boolean =
2787 let
2788   effectSize:Integer = effects->size(),
2789   firstEffect:String = effects->first(),
2790   secondEffectDistance:Integer = effectDistances->at(2).value

```

```

2791 in
2792 subtrace->iterate(
2793   elem:trace::TraceElement;
2794   iter:Tuple(flag:Boolean, i2:Integer, effectCriticalInstant:Integer) = Tuple{flag:Boolean = true, i2:Integer = 1,
      effectCriticalInstant:Integer = 0}
2795   |
2796   let e:String = elem.event in
2797   if e = cause then
2798     Tuple{flag:Boolean = false, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
2799   else
2800     if not iter.flag then
2801       let t:Integer = elem.timestamp in
2802       if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at
          (iter.i2).which) then
2803         if iter.i2 = effectSize then
2804           Tuple{flag:Boolean = true, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
2805         else
2806           let i:Integer = iter.i2 + 1 in
2807           Tuple{flag:Boolean = iter.flag, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).
              value}
2808         endif
2809       else
2810         if e = firstEffect then
2811           Tuple{flag:Boolean = iter.flag, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
2812         else
2813           Tuple{flag:Boolean = iter.flag, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
2814         endif
2815       endif
2816     else
2817       iter
2818     endif
2819   endif
2820 ).flag
2821
2822 =====
2823 def: checkPatternResponseManyManyRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String), effects:
      Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
2824 let
2825   causeSize:Integer = causes->size(),
2826   firstCause:String = causes->first(),
2827   effectSize:Integer = effects->size(),
2828   firstEffect:String = effects->first(),
2829   lastEffect:String = effects->last(),
2830   secondEffectDistance:Integer = effectDistances->at(2).value
2831 in
2832 subtrace->iterate(elem:trace::TraceElement;
2833   iter:Tuple(flag:Boolean, i1:Integer, i2:Integer, effectCriticalInstant:Integer) = Tuple{flag:Boolean = true, i1:
      Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
2834   |
2835   let e:String = elem.event in
2836   if iter.i2 = effectSize and e = lastEffect and self.compare(elem.timestamp, iter.effectCriticalInstant,
      effectDistances->last().which) then
2837     Tuple{flag:Boolean = true, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.
      effectCriticalInstant}
2838   else
2839     if iter.i1 > 1 and e = causes->at(iter.i1) then
2840       if iter.i1 = causeSize then
2841         Tuple{flag:Boolean = false, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.
          effectCriticalInstant}
2842       else
2843         let t:Integer = elem.timestamp in
2844         if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
          at(iter.i2).which) then
2845           let i:Integer = iter.i2 + 1 in
2846           Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = i, effectCriticalInstant:Integer = t
              + effectDistances->at(i).value}
2847         else
2848           if not iter.flag and e = firstEffect then
2849             Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer =

```



```

                t + secondEffectDistance}
2850     else
2851         Tuple{flag:Boolean = iter.flag, i1:Integer = iter.i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer =
                iter.effectCriticalInstant}
2852     endif
2853     endif
2854     endif
2855     else
2856         if e = firstCause then
2857             let t:Integer = elem.timestamp in
2858             if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
                at(iter.i2).which) then
2859                 let i:Integer = iter.i2 + 1 in
2860                 Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = i, effectCriticalInstant:Integer = t +
                effectDistances->at(i).value}
2861             else
2862                 if not iter.flag and e = firstEffect then
2863                     Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = 2, effectCriticalInstant:Integer = t +
                secondEffectDistance}
2864                 else
2865                     Tuple{flag:Boolean = iter.flag, i1:Integer = 2, i2:Integer = 1, effectCriticalInstant:Integer = iter.
                effectCriticalInstant}
2866                 endif
2867             endif
2868         else
2869             let t:Integer = elem.timestamp in
2870             if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
                at(iter.i2).which) then
2871                 let i:Integer = iter.i2 + 1 in
2872                 Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = i, effectCriticalInstant:Integer = t +
                effectDistances->at(i).value}
2873             else
2874                 if not iter.flag and e = firstEffect then
2875                     Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = 2, effectCriticalInstant:Integer = t +
                secondEffectDistance}
2876                 else
2877                     Tuple{flag:Boolean = iter.flag, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.
                effectCriticalInstant}
2878                 endif
2879             endif
2880         endif
2881     endif
2882     endif
2883 ).flag
2884
2885 =====
2886 def: checkPatternResponseOneManyAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:
        Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
2887 let
2888     effectSize:Integer = effects->size(),
2889     firstEffect:String = effects->first(),
2890     secondEffectDistance:Integer = effectDistances->at(2).value
2891 in
2892 subtrace->iterate(elem:trace::TraceElement;
2893     iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer) = Tuple{flag:
        Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
2894 |
2895     let e:String = elem.event in
2896     if e = cause then // latest cause
2897         Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + midDistance, i2:Integer = 1,
            effectCriticalInstant:Integer = iter.effectCriticalInstant}
2898     else
2899         if not iter.flag then
2900             let t:Integer = elem.timestamp in
2901             if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at
                (iter.i2).which) then
2902                 if iter.i2 = effectSize then
2903                     Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1,
                effectCriticalInstant:Integer = iter.effectCriticalInstant}

```

```

2904     else
2905         let i:Integer = iter.i2 + 1 in
2906         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i,
            effectCriticalInstant:Integer = t + effectDistances->at(i).value}
2907     endif
2908 else
2909     if e = firstEffect and t >= iter.midCriticalInstant then
2910         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2,
            effectCriticalInstant:Integer = t + secondEffectDistance}
2911     else
2912         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1,
            effectCriticalInstant:Integer = iter.effectCriticalInstant}
2913     endif
2914 endif
2915 else
2916     iter
2917 endif
2918 endif
2919 ).flag
2920
2921 =====
2922 def: checkPatternResponseOneManyAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:
            Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
2923 let
2924     effectSize:Integer = effects->size(),
2925     firstEffect:String = effects->first(),
2926     secondEffectDistance:Integer = effectDistances->at(2).value
2927 in
2928 subtrace->iterate(elem:trace::TraceElement;
2929     iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
2930 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
2931 |
2932 let e:String = elem.event in
2933 if iter.flag then
2934     if iter.midCriticalInstant = 0 then
2935         if e = cause then
2936             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i2:Integer = 1,
                effectCriticalInstant:Integer = iter.effectCriticalInstant}
2937         else
2938             iter
2939         endif
2940     else
2941         let t:Integer = elem.timestamp in
2942         if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at
            (iter.i2).which) then
2943             if iter.i2 = effectSize then
2944                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:
                    Integer = iter.effectCriticalInstant}
2945             else
2946                 let i:Integer = iter.i2 + 1 in
2947                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i,
                    effectCriticalInstant:Integer = t + effectDistances->at(i).value}
2948             endif
2949         else
2950             if e = firstEffect then
2951                 if t <= iter.midCriticalInstant then
2952                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2,
                        effectCriticalInstant:Integer = t + secondEffectDistance}
2953                 else
2954                     Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i2:Integer = null, effectCriticalInstant:
                        Integer = null}
2955                 endif
2956             else
2957                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 1,
                    effectCriticalInstant:Integer = iter.effectCriticalInstant}
2958             endif
2959         endif
2960     endif
2961 else

```

```

2962     iter
2963   endif
2964 ).midCriticalInstant = 0
2965
2966 =====
2967 def: checkPatternResponseOneManyExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:
      Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
2968 let
2969   effectSize:Integer = effects->size(),
2970   firstEffect:String = effects->first(),
2971   secondEffectDistance:Integer = effectDistances->at(2).value
2972 in
2973 subtrace->iterate(elem:trace::TraceElement;
2974   iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i2:Integer,
      effectCriticalInstant:Integer)
2975   = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i2
      :Integer = 1, effectCriticalInstant:Integer = 0}
2976   |
2977   if iter.flag then
2978     let e:String = elem.event in
2979     if e = cause then
2980       let ct:Integer = elem.timestamp + midDistance in
2981       Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct),
          midCriticalInstant:Integer = ct, i2:Integer = 1, effectCriticalInstant:Integer = iter.
          effectCriticalInstant}
2982     else
2983       let t:Integer = elem.timestamp in
2984       if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
2985         if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
          at(iter.i2).which) then
2986           if iter.i2 = effectSize then
2987             if iter.midCriticalInstants->size() = 1 then
2988               Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
          excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, i2:
          Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
2989             else
2990               let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
2991               Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
          excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, i2:Integer =
          1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
2992             endif
2993           else
2994             let i:Integer = iter.i2 + 1 in
2995             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
          midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = i, effectCriticalInstant:Integer
          = t + effectDistances->at(i).value}
2996           endif
2997         else
2998           if e = firstEffect and t = iter.midCriticalInstant then
2999             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
          midCriticalInstant:Integer = iter.midCriticalInstant, i2:Integer = 2, effectCriticalInstant:Integer
          = t + secondEffectDistance}
3000           else
3001             Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
          midCriticalInstant:Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
3002           endif
3003         endif
3004       else
3005         iter
3006       endif
3007     endif
3008   else
3009     iter
3010   endif
3011 ).midCriticalInstants->isEmpty()
3012
3013 =====
3014 def: checkPatternResponseOneManyMidRight(subtrace:OrderedSet(trace::TraceElement), cause:String, midDistance:TempPsy::
      TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =

```

```

3015 let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator = midDistance.comparingOperator in
3016 if midWhich = TemPsy::ComparingOperator::ATLEAST then
3017   self.checkPatternResponseOneManyAtLeastMidRight(subtrace, cause, midValue, effects, effectDistances)
3018 else
3019   if midWhich = TemPsy::ComparingOperator::ATMOST then
3020     self.checkPatternResponseOneManyAtMostMidRight(subtrace, cause, midValue, effects, effectDistances)
3021   else
3022     self.checkPatternResponseOneManyExactlyMidRight(subtrace, cause, midValue, effects, effectDistances)
3023   endif
3024 endif
3025
3026 =====
3027 def: checkPatternResponseManyManyAtLeastMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):
    Boolean =
3028 let
3029   causeSize:Integer = causes->size(),
3030   firstCause:String = causes->first(),
3031   effectSize:Integer = effects->size(),
3032   firstEffect:String = effects->first(),
3033   lastEffect:String = effects->last(),
3034   secondEffectDistance:Integer = effectDistances->at(2).value
3035 in
3036 subtrace->iterate(elem:trace::TraceElement;
3037   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
3038   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:
    Integer = 0}
3039   |
3040   let e:String = elem.event in
3041   if iter.i2 = effectSize and e = lastEffect and self.compare(elem.timestamp, iter.effectCriticalInstant,
    effectDistances->last().which) then
3042     Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1,
    effectCriticalInstant:Integer = iter.effectCriticalInstant}
3043   else
3044     if iter.i1 > 1 and e = causes->at(iter.i1) then
3045       if iter.i1 = causeSize then
3046         Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1, i2:
    Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3047       else
3048         let t:Integer = elem.timestamp in
3049         if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
    effectDistances->at(iter.i2).which) then
3050           let i:Integer = iter.i2 + 1 in
3051           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1
    + 1, i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
3052         else
3053           if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
3054             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
    i1 + 1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
3055           else
3056             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
    i1 + 1, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3057           endif
3058         endif
3059       endif
3060     else
3061       if e = firstCause then
3062         let t:Integer = elem.timestamp in
3063         if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
    effectDistances->at(iter.i2).which) then
3064           let i:Integer = iter.i2 + 1 in
3065           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
    Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
3066         else
3067           if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
3068             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
    Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
3069           else
3070             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:

```

```

        Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3071     endif
3072     endif
3073     else
3074     let t:Integer = elem.timestamp in
3075     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(elem.timestamp, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
3076     let i:Integer = iter.i2 + 1 in
3077     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
        Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
3078     else
3079     if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
3080     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
        Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
3081     else
3082     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
        Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3083     endif
3084     endif
3085     endif
3086     endif
3087     endif
3088 ).flag
3089
3090 =====
3091 def: checkPatternResponseManyManyAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
        midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):
        Boolean =
3092 let
3093 causeSize:Integer = causes->size(),
3094 firstCause:String = causes->first(),
3095 effectSize:Integer = effects->size(),
3096 firstEffect:String = effects->first(),
3097 secondEffectDistance:Integer = effectDistances->at(2).value
3098 in
3099 subtrace->iterate(elem:trace::TraceElement;
3100 iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, i2:Integer, effectCriticalInstant:Integer)
3101 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, i2:Integer = 1, effectCriticalInstant:
        Integer = 0}
3102 |
3103 let e:String = elem.event in
3104 if iter.flag then
3105     if iter.midCriticalInstant = 0 then
3106     if iter.i1 > 1 and e = causes->at(iter.i1) then
3107     if iter.i1 = causeSize then
3108     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
        i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3109     else
3110     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1
        + 1, i2:Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3111     endif
3112     else
3113     if e = firstCause then
3114     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:
        Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3115     else
3116     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:
        Integer = iter.i2, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3117     endif
3118     endif
3119     else
3120     let t:Integer = elem.timestamp in
3121     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at
        (iter.i2).which) then
3122     if iter.i2 = effectSize then
3123     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = iter.i1, i2:Integer = 1,
        effectCriticalInstant:Integer = iter.effectCriticalInstant}
3124     else
3125     let i:Integer = iter.i2 + 1 in

```

```

3126     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1,
        i2:Integer = i, effectCriticalInstant:Integer = t + effectDistances->at(i).value}
3127     endif
3128     else
3129     if e = firstEffect then
3130     if t <= iter.midCriticalInstant then
3131     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
        i1, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
3132     else
3133     Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, i2:Integer = null,
        effectCriticalInstant:Integer = null}
3134     endif
3135     else
3136     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1,
        i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3137     endif
3138     endif
3139     endif
3140     else
3141     iter
3142     endif
3143 ) .midCriticalInstant = 0
3144
3145 =====
3146 def: checkPatternResponseManyManyExactlyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
        midDistance:Integer, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:Integer))):
        Boolean =
3147 let
3148 causeSize:Integer = causes->size(),
3149 firstCause:String = causes->first(),
3150 effectSize:Integer = effects->size(),
3151 firstEffect:String = effects->first(),
3152 lastEffect:String = effects->last(),
3153 secondEffectDistance:Integer = effectDistances->at(2).value
3154 in
3155 subtrace->iterate(elem:trace::TraceElement;
3156 iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer, i2:Integer,
        effectCriticalInstant:Integer)
3157 = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1
        :Integer = 1, i2:Integer = 1, effectCriticalInstant:Integer = 0}
3158 |
3159 if iter.flag then
3160 let e:String = elem.event, t:Integer = elem.timestamp in
3161 if iter.i2 = effectSize and e = lastEffect then
3162 if iter.midCriticalInstants->size() = 1 then
3163 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}
3164 else
3165 let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
3166 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}
3167 endif
3168 else
3169 if iter.i1 > 1 and e = causes->at(iter.i1) then
3170 if iter.i1 = causeSize then
3171 let ct:Integer = t + midDistance in
3172 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}
3173 else
3174 if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3175 if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
3176 let i:Integer = iter.i2 + 1 in
3177 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}

```

```

3178     else
3179         if e = firstEffect and t = iter.midCriticalInstant then
3180             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}
3181         else
3182             Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                 midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null, effectCriticalInstant:
                 Integer = null}
3183         endif
3184     endif
3185     else
3186         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}
3187     endif
3188 endif
3189 else
3190     if e = firstCause then
3191         if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3192             if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
                 effectDistances->at(iter.i2).which) then
3193                 let i:Integer = iter.i2 + 1 in
3194                 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}
3195             else
3196                 if e = firstEffect and t = iter.midCriticalInstant then
3197                     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                         midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 2,
                         effectCriticalInstant:Integer = t + secondEffectDistance}
3198                 else
3199                     Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                         midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null, effectCriticalInstant:
                         Integer = null}
3200                 endif
3201             endif
3202         else
3203             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                 midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, i2:Integer = 1,
                 effectCriticalInstant:Integer = iter.effectCriticalInstant}
3204         endif
3205     else
3206         if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3207             if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
                 effectDistances->at(iter.i2).which) then
3208                 let i:Integer = iter.i2 + 1 in
3209                 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}
3210             else
3211                 if e = firstEffect and t = iter.midCriticalInstant then
3212                     Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                         midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 2,
                         effectCriticalInstant:Integer = t + secondEffectDistance}
3213                 else
3214                     Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                         midCriticalInstant:Integer = null, i1:Integer = null, i2:Integer = null, effectCriticalInstant:
                         Integer = null}
3215                 endif
3216             endif
3217         else
3218             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                 midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, i2:Integer = 1,
                 effectCriticalInstant:Integer = iter.effectCriticalInstant}
3219         endif
3220     endif
3221 endif
3222 endif

```

```

3223 else
3224   iter
3225 endif
3226 ).midCriticalInstants->isEmpty()
3227
3228 =====
3229 def: checkPatternResponseManyManyMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      midDistance:TempPsy::TimeDistance, effects:Sequence(String), effectDistances:Sequence(Tuple(which:Integer, value:
      Integer))):Boolean =
3230 let midValue:Integer = midDistance.value, midWhich:TempPsy::ComparingOperator = midDistance.comparingOperator in
3231 if midWhich = TempPsy::ComparingOperator::ATLEAST then
3232   self.checkPatternResponseManyManyAtLeastMidRight(subtrace, causes, midValue, effects, effectDistances)
3233 else
3234   if midWhich = TempPsy::ComparingOperator::ATMOST then
3235     self.checkPatternResponseManyManyAtMostMidRight(subtrace, causes, midValue, effects, effectDistances)
3236   else
3237     self.checkPatternResponseManyManyExactlyMidRight(subtrace, causes, midValue, effects, effectDistances)
3238   endif
3239 endif
3240
3241 =====
3242 def: checkPatternResponseManyOneLeft(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effect:String):Boolean =
3243 let
3244   causeSize:Integer = causes->size(),
3245   firstCause:String = causes->first(),
3246   secondCauseDistance:Integer = causeDistances->at(2).value
3247 in
3248 subtrace->iterate(elem:trace::TraceElement;
3249   iter:Tuple(flag:Boolean, i1:Integer, causeCriticalInstant:Integer) = Tuple{flag:Boolean = true, i1:Integer = 1,
      causeCriticalInstant:Integer = 0}
3250 |
3251   let e:String = elem.event in
3252   if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
      causeDistances->at(iter.i1).which) then
3253     if iter.i1 = causeSize then
3254       Tuple{flag:Boolean = false, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
3255     else
3256       let i:Integer = iter.i1 + 1 in
3257       Tuple{flag:Boolean = iter.flag, i1:Integer = i, causeCriticalInstant:Integer = elem.timestamp + causeDistances
        ->at(i).value}
3258     endif
3259   else
3260     if e = firstCause then
3261       Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp +
        secondCauseDistance}
3262     else
3263       if e = effect then
3264         Tuple{flag:Boolean = true, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
3265       else
3266         Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
3267       endif
3268     endif
3269   endif
3270 ).flag
3271
3272 =====
3273 def: checkPatternResponseManyManyLeft(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), effects:Sequence(String)):Boolean =
3274 let
3275   causeSize:Integer = causes->size(),
3276   firstCause:String = causes->first(),
3277   secondCauseDistance:Integer = causeDistances->at(2).value,
3278   effectSize:Integer = effects->size(),
3279   firstEffect:String = effects->first(),
3280   lastEffect:String = effects->last()
3281 in
3282 subtrace->iterate(elem:trace::TraceElement;
3283   iter:Tuple(flag:Boolean, i1:Integer, causeCriticalInstant:Integer, i2:Integer) = Tuple{flag:Boolean = true, i1:

```



```

Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1}
3284 |
3285 let e:String = elem.event in
3286 if iter.i2 = effectSize and e = lastEffect then
3287   Tuple{flag:Boolean = true, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer =
      1}
3288 else
3289   if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
      causeDistances->at(iter.i1).which) then
3290     if iter.i1 = causeSize then
3291       Tuple{flag:Boolean = false, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:
          Integer = 1}
3292     else
3293       let i:Integer = iter.i1 + 1 in
3294       if iter.i2 > 1 and e = effects->at(iter.i2) then
3295         Tuple{flag:Boolean = iter.flag, i1:Integer = i, causeCriticalInstant:Integer = elem.timestamp +
            causeDistances->at(i).value, i2:Integer = iter.i2 + 1}
3296       else
3297         if not iter.flag and e = firstEffect then
3298           Tuple{flag:Boolean = iter.flag, i1:Integer = i, causeCriticalInstant:Integer = elem.timestamp +
              causeDistances->at(i).value, i2:Integer = 2}
3299         else
3300           Tuple{flag:Boolean = iter.flag, i1:Integer = i, causeCriticalInstant:Integer = elem.timestamp +
              causeDistances->at(i).value, i2:Integer = 1}
3301         endif
3302       endif
3303     endif
3304   else
3305     if e = firstCause then
3306       if iter.i2 > 1 and e = effects->at(iter.i2) then
3307         Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp +
            secondCauseDistance, i2:Integer = iter.i2 + 1}
3308       else
3309         if not iter.flag and e = firstEffect then
3310           Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp +
              secondCauseDistance, i2:Integer = 2}
3311         else
3312           Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp +
              secondCauseDistance, i2:Integer = 1}
3313         endif
3314       endif
3315     else
3316       if iter.i2 > 1 and e = effects->at(iter.i2) then
3317         Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
            i2:Integer = iter.i2 + 1}
3318       else
3319         if not iter.flag and e = firstEffect then
3320           Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
              i2:Integer = 2}
3321         else
3322           Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
              i2:Integer = 1}
3323         endif
3324       endif
3325     endif
3326   endif
3327 endif
3328 ).flag
3329
3330 =====
3331 def: checkPatternResponseManyOneLeftAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
3332 let
3333   causeSize:Integer = causes->size(),
3334   firstCause:String = causes->first(),
3335   secondCauseDistance:Integer = causeDistances->at(2).value
3336 in
3337 subtrace->iterate(elem:trace::TraceElement;
3338   iter:Tuple(midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer) = Tuple{midCriticalInstant:Integer

```

```

    = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
3339 |
3340 let e:String = elem.event, t:Integer = elem.timestamp in
3341 if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(iter.
    i1).which) then
3342   if iter.i1 = causeSize then
3343     Tuple{midCriticalInstant:Integer = t + midDistance, i1:Integer = 1, causeCriticalInstant:Integer = iter.
        causeCriticalInstant}
3344   else
3345     let i:Integer = iter.i1 + 1 in
3346     Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i, causeCriticalInstant:Integer = t +
        causeDistances->at(i).value}
3347   endif
3348 else
3349   if e = firstCause then
3350     Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2, causeCriticalInstant:Integer = t +
        secondCauseDistance}
3351   else
3352     if e = effect and t >= iter.midCriticalInstant then
3353       Tuple{midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = iter.
        causeCriticalInstant}
3354     else
3355       Tuple{midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer =
        iter.causeCriticalInstant}
3356     endif
3357   endif
3358 endif
3359 ).midCriticalInstant = 0
3360
3361 =====
3362 def: checkPatternResponseManyOneLeftAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
3363 let
3364   causeSize:Integer = causes->size(),
3365   firstCause:String = causes->first(),
3366   secondCauseDistance:Integer = causeDistances->at(2).value
3367 in
3368 subtrace->iterate(elem:trace::TraceElement;
3369   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer)
3370 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0}
3371 |
3372 let e:String = elem.event in
3373 if iter.flag then
3374   if iter.midCriticalInstant = 0 then
3375     let t:Integer = elem.timestamp in
3376     if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(
        iter.i1).which) then
3377       if iter.i1 = causeSize then
3378         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = t + midDistance, i1:Integer = 1,
            causeCriticalInstant:Integer = iter.causeCriticalInstant}
3379       else
3380         let i:Integer = iter.i1 + 1 in
3381         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i,
            causeCriticalInstant:Integer = t + causeDistances->at(i).value}
3382       endif
3383     else
3384       if e = firstCause then
3385         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
            causeCriticalInstant:Integer = t + secondCauseDistance}
3386       else
3387         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
            causeCriticalInstant:Integer = iter.causeCriticalInstant}
3388       endif
3389     endif
3390   else
3391     if e = effect then
3392       if elem.timestamp <= iter.midCriticalInstant then
3393         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:
            Integer = iter.causeCriticalInstant}

```

```

3394     else
3395         Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, il:Integer = null, causeCriticalInstant:
           Integer = null}
3396     endif
3397     else
3398         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, il:Integer = 1,
           causeCriticalInstant:Integer = iter.causeCriticalInstant}
3399     endif
3400     endif
3401     else
3402         iter
3403     endif
3404 ).midCriticalInstant = 0
3405
3406 =====
3407 def: checkPatternResponseManyOneLeftExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
           causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effect:String):Boolean =
3408 let
3409     causeSize:Integer = causes->size(),
3410     firstCause:String = causes->first(),
3411     secondCauseDistance:Integer = causeDistances->at(2).value
3412 in
3413 subtrace->iterate(elem:trace::TraceElement;
3414     iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, il:Integer,
           causeCriticalInstant:Integer)
3415 = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, il
           :Integer = 1, causeCriticalInstant:Integer = 0}
3416 |
3417 if iter.flag then
3418     let e:String = elem.event, t:Integer = elem.timestamp in
3419     if iter.il > 1 and e = causes->at(iter.il) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(iter
           .il).which) then
3420         if iter.il = causeSize then
3421             let ct:Integer = t + midDistance in
3422             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->append(ct),
           midCriticalInstant:Integer = ct, il:Integer = 1, causeCriticalInstant:Integer = iter.
           causeCriticalInstant}
3423         else
3424             let i:Integer = iter.il + 1 in
3425             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
           midCriticalInstant:Integer = iter.midCriticalInstant, il:Integer = i, causeCriticalInstant:Integer = t +
           causeDistances->at(i).value}
3426         endif
3427     else
3428         if e = firstCause then
3429             Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
           midCriticalInstant:Integer = iter.midCriticalInstant, il:Integer = 2, causeCriticalInstant:Integer = t +
           secondCauseDistance}
3430         else
3431             if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3432                 if t = iter.midCriticalInstant and e = effect then
3433                     if iter.midCriticalInstants->size() = 1 then
3434                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
           excluding(iter.midCriticalInstant), midCriticalInstant:Integer = iter.midCriticalInstant, il:
           Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
3435                     else
3436                         let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
3437                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants->
           excluding(iter.midCriticalInstant), midCriticalInstant:Integer = nextCriticalInstant, il:Integer =
           1, causeCriticalInstant:Integer = iter.causeCriticalInstant}
3438                     endif
3439                 else
3440                     Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
           midCriticalInstant:Integer = null, il:Integer = null, causeCriticalInstant:Integer = null}
3441                 endif
3442             else
3443                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
           midCriticalInstant:Integer = iter.midCriticalInstant, il:Integer = 1, causeCriticalInstant:Integer =
           iter.causeCriticalInstant}

```

```

3444     endif
3445   endif
3446   endif
3447   else
3448     iter
3449   endif
3450 ).midCriticalInstants->isEmpty()
3451
3452 =====
3453 def: checkPatternResponseManyOneLeftMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TemPsy::TimeDistance, effect:String):
      Boolean =
3454 let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator = midDistance.comparingOperator in
3455 if midWhich = TemPsy::ComparingOperator::ATLEAST then
3456   self.checkPatternResponseManyOneLeftAtLeastMid(subtrace, causes, causeDistances, midValue, effect)
3457 else
3458   if midWhich = TemPsy::ComparingOperator::ATMOST then
3459     self.checkPatternResponseManyOneLeftAtMostMid(subtrace, causes, causeDistances, midValue, effect)
3460   else
3461     self.checkPatternResponseManyOneLeftExactlyMid(subtrace, causes, causeDistances, midValue, effect)
3462   endif
3463 endif
3464
3465 =====
3466 def: checkPatternResponseManyManyLeftAtLeastMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)):
      Boolean =
3467 let
3468   causeSize:Integer = causes->size(),
3469   firstCause:String = causes->first(),
3470   secondCauseDistance:Integer = causeDistances->at(2).value,
3471   effectSize:Integer = effects->size(),
3472   firstEffect:String = effects->first(),
3473   lastEffect:String = effects->last()
3474 in
3475 subtrace->iterate(elem:trace::TraceElement;
3476   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer)
3477   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
      Integer = 1}
3478   |
3479   let e:String = elem.event in
3480   if iter.i2 = effectSize and e = lastEffect then
3481     Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
      causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
3482   else
3483     if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
      causeDistances->at(iter.i1).which) then
3484       if iter.i1 = causeSize then
3485         Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
          causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
3486       else
3487         let i:Integer = iter.i1 + 1 in
3488         if iter.i2 > 1 and e = effects->at(iter.i2) then
3489           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}
3490         else
3491           if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
3492             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i,
              causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = 2}
3493           else
3494             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i,
              causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = 1}
3495           endif
3496         endif
3497       endif
3498     else
3499       if e = firstCause then
3500         if iter.i2 > 1 and e = effects->at(iter.i2) then
3501           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,

```

```

        causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2 + 1}
3502     else
3503         if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
3504             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 2}
3505         else
3506             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = 1}
3507         endif
3508     endif
3509     else
3510         if iter.i2 > 1 and e = effects->at(iter.i2) then
3511             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
3512         else
3513             if not iter.flag and e = firstEffect and elem.timestamp >= iter.midCriticalInstant then
3514                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                    causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
3515             else
3516                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                    causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
3517             endif
3518         endif
3519     endif
3520 endif
3521 endif
3522 ).flag
3523
3524 =====
3525 def: checkPatternResponseManyManyLeftAtMostMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
        causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)):
        Boolean =
3526 let
3527     causeSize:Integer = causes->size(),
3528     firstCause:String = causes->first(),
3529     secondCauseDistance:Integer = causeDistances->at(2).value,
3530     effectSize:Integer = effects->size(),
3531     firstEffect:String = effects->first()
3532 in
3533 subtrace->iterate(elem:trace::TraceElement;
3534     iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer)
3535     = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
        Integer = 1} |
3536     let e:String = elem.event in
3537     if iter.flag then
3538         if iter.midCriticalInstant = 0 then
3539             if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
                causeDistances->at(iter.i1).which) then
3540                 if iter.i1 = causeSize then
3541                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
                        causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2}
3542                 else
3543                     let i:Integer = iter.i1 + 1 in
3544                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i,
                        causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i).value, i2:Integer = iter.i2}
3545                 endif
3546             else
3547                 if e = firstCause then
3548                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
                        causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2}
3549                 else
3550                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
                        causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2}
3551                 endif
3552             endif
3553         else
3554             if iter.i2 > 1 and e = effects->at(iter.i2) then
3555                 if iter.i2 = effectSize then
3556                     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = iter.i1, causeCriticalInstant:

```

```

        Integer = iter.causeCriticalInstant, i2:Integer = 1}
3557     else
3558         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1,
            causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2 + 1}
3559     endif
3560     else
3561         if e = firstEffect then
3562             if elem.timestamp <= iter.midCriticalInstant then
3563                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
                    i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2}
3564             else
3565                 Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:
                    Integer = null, i2:Integer = null}
3566             endif
3567         else
3568             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1,
                causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1}
3569         endif
3570     endif
3571 endif
3572 else
3573     iter
3574 endif
3575 ).midCriticalInstant = 0
3576
3577 =====
3578 def: checkPatternResponseManyManyLeftExactlyMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
    causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String)):
    Boolean =
3579 let
3580     causeSize:Integer = causes->size(),
3581     firstCause:String = causes->first(),
3582     secondCauseDistance:Integer = causeDistances->at(2).value,
3583     effectSize:Integer = effects->size(),
3584     firstEffect:String = effects->first(),
3585     lastEffect:String = effects->last()
3586 in
3587 subtrace->iterate(elem:trace::TraceElement;
3588     iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer,
        causeCriticalInstant:Integer, i2:Integer)
3589 = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1
    :Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1} |
3590 if iter.flag then
3591     let e:String = elem.event, t:Integer = elem.timestamp in
3592     if iter.i2 = effectSize and e = lastEffect then
3593         if iter.midCriticalInstants->size() = 1 then
3594             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}
3595         else
3596             let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
3597             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}
3598         endif
3599     else
3600         if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(
            iter.i1).which) then
3601             if iter.i1 = causeSize then
3602                 let ct:Integer = t + midDistance in
3603                 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}
3604             else
3605                 let i:Integer = iter.i1 + 1 in
3606                 if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3607                     if iter.i2 > 1 and e = effects->at(iter.i2) then
3608                         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}
3609     else
3610         if e = firstEffect and t = iter.midCriticalInstant then
3611             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}
3612         else
3613             Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                 midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                 Integer = null}
3614         endif
3615     endif
3616     else
3617         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}
3618     endif
3619     endif
3620     else
3621         if e = firstCause then
3622             if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3623                 if iter.i2 > 1 and e = effects->at(iter.i2) then
3624                     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}
3625                 else
3626                     if e = firstEffect and t = iter.midCriticalInstant then
3627                         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}
3628                     else
3629                         Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                                 midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                                 Integer = null}
3630                     endif
3631                 endif
3632             else
3633                 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}
3634             endif
3635         else
3636             if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3637                 if iter.i2 > 1 and e = effects->at(iter.i2) then
3638                     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}
3639                 else
3640                     if e = firstEffect and t = iter.midCriticalInstant then
3641                         Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                                 midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:
                                 Integer = iter.causeCriticalInstant, i2:Integer = 2}
3642                     else
3643                         Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                                 midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                                 Integer = null}
3644                     endif
3645                 endif
3646             else
3647                 Tuple{flag:Boolean = iter.flag, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                         midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1, causeCriticalInstant:Integer =
                         iter.causeCriticalInstant, i2:Integer = 1}
3648             endif
3649         endif
3650     endif
3651     endif
3652     else
3653         iter
3654     endif

```

```

3655 ).midCriticalInstants->isEmpty()
3656
3657 =====
3658 def: checkPatternResponseManyManyLeftMid(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String),
      causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:TempPsy::TimeDistance, effects:Sequence
      (String)):Boolean =
3659 let midValue:Integer = midDistance.value, midWhich:TempPsy::ComparingOperator = midDistance.comparingOperator in
3660 if midWhich = TempPsy::ComparingOperator::ATLEAST then
3661   self.checkPatternResponseManyManyLeftAtLeastMid(subtrace, causes, causeDistances, midValue, effects)
3662 else
3663   if midWhich = TempPsy::ComparingOperator::ATMOST then
3664     self.checkPatternResponseManyManyLeftAtMostMid(subtrace, causes, causeDistances, midValue, effects)
3665   else
3666     self.checkPatternResponseManyManyLeftExactlyMid(subtrace, causes, causeDistances, midValue, effects)
3667   endif
3668 endif
3669
3670 =====
3671 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 =
3672 let
3673   causeSize:Integer = causes->size(),
3674   firstCause:String = causes->first(),
3675   secondCauseDistance:Integer = causeDistances->at(2).value,
3676   effectSize:Integer = effects->size(),
3677   firstEffect:String = effects->first(),
3678   lastEffect:String = effects->last(),
3679   secondEffectDistance:Integer = effectDistances->at(2).value
3680 in
3681 subtrace->iterate(elem:trace::TraceElement;
3682   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}
3683 |
3684 let e:String = elem.event in
3685 if iter.i2 = effectSize and e = lastEffect and self.compare(elem.timestamp, iter.effectCriticalInstant,
      effectDistances->last().which) then
3686   Tuple{flag:Boolean = true, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer =
      1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3687 else
3688   if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
      causeDistances->at(iter.i1).which) then
3689     if iter.i1 = causeSize then
3690       Tuple{flag:Boolean = false, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:
      Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3691     else
3692       let t:Integer = elem.timestamp, i11:Integer = iter.i1 + 1 in
3693       if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
      at(iter.i2).which) then
3694         let i22:Integer = iter.i2 + 1 in
3695         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}
3696       else
3697         if not iter.flag and e = firstEffect then
3698           Tuple{flag:Boolean = iter.flag, i1:Integer = i11, causeCriticalInstant:Integer = t + causeDistances->at(
      i11).value, i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
3699         else
3700           Tuple{flag:Boolean = iter.flag, i1:Integer = i11, causeCriticalInstant:Integer = elem.timestamp +
      causeDistances->at(i11).value, i2:Integer = 1, effectCriticalInstant:Integer = iter.
      effectCriticalInstant}
3701         endif
3702       endif
3703     endif
3704   else
3705     if e = firstCause then
3706       let t:Integer = elem.timestamp in
3707       if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
      at(iter.i2).which) then

```



```

3708     let i22:Integer = iter.i2 + 1 in
3709     Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = t + secondCauseDistance, i2:
          Integer = i22, effectCriticalInstant:Integer = t + causeDistances->at(i22).value}
3710 else
3711     if not iter.flag and e = firstEffect then
3712         Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = t + secondCauseDistance,
          i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
3713     else
3714         Tuple{flag:Boolean = iter.flag, i1:Integer = 2, causeCriticalInstant:Integer = elem.timestamp +
          secondCauseDistance, i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3715     endif
3716 endif
3717 else
3718     let t:Integer = elem.timestamp in
3719     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
          at(iter.i2).which) then
3720         let i22:Integer = iter.i2 + 1 in
3721         Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
          i2:Integer = i22, effectCriticalInstant:Integer = t + causeDistances->at(i22).value}
3722     else
3723         if not iter.flag and e = firstEffect then
3724             Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
          i2:Integer = 2, effectCriticalInstant:Integer = t + secondEffectDistance}
3725         else
3726             Tuple{flag:Boolean = iter.flag, i1:Integer = 1, causeCriticalInstant:Integer = iter.causeCriticalInstant,
          i2:Integer = 1, effectCriticalInstant:Integer = iter.effectCriticalInstant}
3727         endif
3728     endif
3729 endif
3730 endif
3731 endif
3732 ).flag
3733
3734
3735 =====
3736 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 =
3737 let
3738     causeSize:Integer = causes->size(),
3739     firstCause:String = causes->first(),
3740     secondCauseDistance:Integer = causeDistances->at(2).value,
3741     effectSize:Integer = effects->size(),
3742     firstEffect:String = effects->first(),
3743     lastEffect:String = effects->last(),
3744     secondEffectDistance:Integer = effectDistances->at(2).value
3745 in
3746 subtrace->iterate(elem:trace::TraceElement;
3747     iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer,
          effectCriticalInstant:Integer)
3748 = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
          Integer = 1, effectCriticalInstant:Integer = 0}
3749 |
3750 let e:String = elem.event in
3751 if iter.i2 = effectSize and e = lastEffect and self.compare(elem.timestamp, iter.effectCriticalInstant,
          effectDistances->last().which) then
3752     Tuple{flag:Boolean = true, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
          causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer =
          iter.effectCriticalInstant}
3753 else
3754     if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
          causeDistances->at(iter.i1).which) then
3755         if iter.i1 = causeSize then
3756             Tuple{flag:Boolean = false, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
          causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer
          = iter.effectCriticalInstant}
3757         else
3758             let t:Integer = elem.timestamp, i11:Integer = iter.i1 + 1 in
3759             if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->

```

```

3760         at(iter.i2).which) then
3761     let i22:Integer = iter.i2 + 1 in
3762     Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
3763         causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = i22,
3764         effectCriticalInstant:Integer = t + causeDistances->at(i22).value}
3765 else
3766     if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
3767         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
3768             causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = 2,
3769             effectCriticalInstant:Integer = t + secondEffectDistance}
3770     else
3771         Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
3772             causeCriticalInstant:Integer = t + causeDistances->at(i11).value, i2:Integer = 1,
3773             effectCriticalInstant:Integer = iter.effectCriticalInstant}
3774     endif
3775 endif
3776 else
3777     if e = firstCause then
3778         let t:Integer = elem.timestamp in
3779         if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
3780             at(iter.i2).which) then
3781             let i22:Integer = iter.i2 + 1 in
3782             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
3783                 causeCriticalInstant:Integer = t + secondCauseDistance, i2:Integer = i22, effectCriticalInstant:
3784                 Integer = elem.timestamp + causeDistances->at(i22).value}
3785         else
3786             if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
3787                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
3788                     causeCriticalInstant:Integer = t + secondCauseDistance, i2:Integer = 2, effectCriticalInstant:
3789                     Integer = elem.timestamp + secondEffectDistance}
3790             else
3791                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
3792                     causeCriticalInstant:Integer = t + secondCauseDistance, i2:Integer = 1, effectCriticalInstant:
3793                     Integer = iter.effectCriticalInstant}
3794             endif
3795         endif
3796     else
3797         let t:Integer = elem.timestamp in
3798         if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->
3799             at(iter.i2).which) then
3800             let i22:Integer = iter.i2 + 1 in
3801             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
3802                 causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:
3803                 Integer = elem.timestamp + causeDistances->at(i22).value}
3804         else
3805             if not iter.flag and e = firstEffect and t >= iter.midCriticalInstant then
3806                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
3807                     causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:
3808                     Integer = elem.timestamp + secondEffectDistance}
3809             else
3810                 Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
3811                     causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:
3812                     Integer = iter.effectCriticalInstant}
3813             endif
3814         endif
3815     endif
3816 endif
3817 ) .flag
3818 =====
3819 def: checkPatternResponseManyManyLeftAtMostMidRight(subtrace:OrderedSet(trace::TraceElement), causes:Sequence(String)
3820     , causeDistances:Sequence(Tuple(which:Integer, value:Integer)), midDistance:Integer, effects:Sequence(String),
3821     effectDistances:Sequence(Tuple(which:Integer, value:Integer))):Boolean =
3822 let
3823     causeSize:Integer = causes->size(),
3824     firstCause:String = causes->first(),
3825     secondCauseDistance:Integer = causeDistances->at(2).value,

```

```

3806 effectSize:Integer = effects->size(),
3807 firstEffect:String = effects->first(),
3808 secondEffectDistance:Integer = effectDistances->at(2).value
3809 in
3810 subtrace->iterate(elem:trace::TraceElement;
3811   iter:Tuple(flag:Boolean, midCriticalInstant:Integer, i1:Integer, causeCriticalInstant:Integer, i2:Integer,
3812     effectCriticalInstant:Integer)
3813   = Tuple{flag:Boolean = true, midCriticalInstant:Integer = 0, i1:Integer = 1, causeCriticalInstant:Integer = 0, i2:
3814     Integer = 1, effectCriticalInstant:Integer = 0}
3815   |
3816   let e:String = elem.event in
3817   if iter.flag then
3818     if iter.midCriticalInstant = 0 then
3819       if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(elem.timestamp, iter.causeCriticalInstant,
3820         causeDistances->at(iter.i1).which) then
3821         if iter.i1 = causeSize then
3822           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = elem.timestamp + midDistance, i1:Integer = 1,
3823             causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2, effectCriticalInstant:
3824               Integer = iter.effectCriticalInstant}
3825         else
3826           let i11:Integer = iter.i1 + 1 in
3827           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = i11,
3828             causeCriticalInstant:Integer = elem.timestamp + causeDistances->at(i11).value, i2:Integer = iter.i2,
3829             effectCriticalInstant:Integer = iter.effectCriticalInstant}
3830         endif
3831       else
3832         if e = firstCause then
3833           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 2,
3834             causeCriticalInstant:Integer = elem.timestamp + secondCauseDistance, i2:Integer = iter.i2,
3835             effectCriticalInstant:Integer = iter.effectCriticalInstant}
3836         else
3837           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = 1,
3838             causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = iter.i2, effectCriticalInstant:
3839               Integer = iter.effectCriticalInstant}
3840         endif
3841       endif
3842     else
3843       let t:Integer = elem.timestamp in
3844       if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant, effectDistances->at
3845         (iter.i2).which) then
3846         if iter.i2 = effectSize then
3847           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = 0, i1:Integer = iter.i1, causeCriticalInstant:
3848             Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:Integer = iter.
3849               effectCriticalInstant}
3850         else
3851           let i22:Integer = iter.i2 + 1 in
3852           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1,
3853             causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = i22, effectCriticalInstant:
3854               Integer = t + effectDistances->at(i22).value}
3855         endif
3856       else
3857         if e = firstEffect then
3858           if t <= iter.midCriticalInstant then
3859             Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.
3860               i1, causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 2, effectCriticalInstant:
3861                 Integer = t + effectDistances->at(2).value}
3862           else
3863             Tuple{flag:Boolean = false, midCriticalInstant:Integer = -1, i1:Integer = null, causeCriticalInstant:
3864               Integer = null, i2:Integer = null, effectCriticalInstant:Integer = null}
3865           endif
3866         else
3867           Tuple{flag:Boolean = iter.flag, midCriticalInstant:Integer = iter.midCriticalInstant, i1:Integer = iter.i1,
3868             causeCriticalInstant:Integer = iter.causeCriticalInstant, i2:Integer = 1, effectCriticalInstant:
3869               Integer = iter.effectCriticalInstant}
3870         endif
3871       endif
3872     endif
3873   else
3874     iter

```

```

3854 endif
3855 ).midCriticalInstant = 0
3856
3857 =====
3858 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 =
3859 let
3860   causeSize:Integer = causes->size(),
3861   firstCause:String = causes->first(),
3862   secondCauseDistance:Integer = causeDistances->at(2).value,
3863   effectSize:Integer = effects->size(),
3864   firstEffect:String = effects->first(),
3865   lastEffect:String = effects->last(),
3866   secondEffectDistance:Integer = effectDistances->at(2).value
3867 in
3868 subtrace->iterate(elem:trace::TraceElement;
3869   iter:Tuple(flag:Boolean, midCriticalInstants:Sequence(Integer), midCriticalInstant:Integer, i1:Integer,
    causeCriticalInstant:Integer, i2:Integer, effectCriticalInstant:Integer)
3870 = Tuple{flag:Boolean = true, midCriticalInstants:Sequence(Integer) = Sequence{}, midCriticalInstant:Integer = 0, i1
    :Integer = 1, causeCriticalInstant:Integer = 0, i2:Integer = 1, effectCriticalInstant:Integer = 0}
3871 |
3872 if iter.flag then
3873   let e:String = elem.event, t:Integer = elem.timestamp in
3874   if iter.i2 = effectSize and e = lastEffect then
3875     if iter.midCriticalInstants->size() = 1 then
3876       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}
3877     else
3878       let nextCriticalInstant:Integer = iter.midCriticalInstants->at(2) in
3879       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}
3880     endif
3881   else
3882     if iter.i1 > 1 and e = causes->at(iter.i1) and self.compare(t, iter.causeCriticalInstant, causeDistances->at(
        iter.i1).which) then
3883       if iter.i1 = causeSize then
3884         let ct:Integer = t + midDistance in
3885         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}
3886       else
3887         let i11:Integer = iter.i1 + 1 in
3888         if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3889           if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
            effectDistances->at(iter.i2).which) then
3890             let i22:Integer = iter.i2 + 1 in
3891             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}
3892           else
3893             if e = firstEffect and t = iter.midCriticalInstant then
3894               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}
3895             else
3896               Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
                midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
                Integer = null, effectCriticalInstant:Integer = null}
3897             endif
3898           endif
3899         else
3900           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}
3901     endif
3902     endif
3903     else
3904     if e = firstCause then
3905     if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3906     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
3907     let i22:Integer = iter.i2 + 1 in
3908     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}
3909     else
3910     if e = firstEffect and t = iter.midCriticalInstant then
3911     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}
3912     else
3913     Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
        midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
        Integer = null, effectCriticalInstant:Integer = null}
3914     endif
3915     endif
3916     else
3917     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}
3918     endif
3919     else
3920     if iter.midCriticalInstants->notEmpty() and t >= iter.midCriticalInstant then
3921     if iter.i2 > 1 and e = effects->at(iter.i2) and self.compare(t, iter.effectCriticalInstant,
        effectDistances->at(iter.i2).which) then
3922     let i22:Integer = iter.i2 + 1 in
3923     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}
3924     else
3925     if e = firstEffect and t = iter.midCriticalInstant then
3926     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}
3927     else
3928     Tuple{flag:Boolean = false, midCriticalInstants:Sequence(Integer) = iter.midCriticalInstants,
        midCriticalInstant:Integer = null, i1:Integer = null, causeCriticalInstant:Integer = null, i2:
        Integer = null, effectCriticalInstant:Integer = null}
3929     endif
3930     endif
3931     else
3932     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}
3933     endif
3934     endif
3935     endif
3936     endif
3937     else
3938     iter
3939     endif
3940 )midCriticalInstants->isEmpty()
3941
3942 =====

```

```
3943 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 =
3944 let midValue:Integer = midDistance.value, midWhich:TemPsy::ComparingOperator = midDistance.comparingOperator in
3945 if midWhich = TemPsy::ComparingOperator::ATLEAST then
3946     self.checkPatternResponseManyManyLeftAtLeastMidRight(subtrace, causes, causeDistances, midValue, effects,
        effectDistances)
3947 else
3948     if midWhich = TemPsy::ComparingOperator::ATMOST then
3949         self.checkPatternResponseManyManyLeftAtMostMidRight(subtrace, causes, causeDistances, midValue, effects,
            effectDistances)
3950     else
3951         self.checkPatternResponseManyManyLeftExactlyMidRight(subtrace, causes, causeDistances, midValue, effects,
            effectDistances)
3952     endif
3953 endif
```
