Noname manuscript No.
(will be inserted by the editor)

An Expressive Event-based Language for Representing User Behavior Patterns

Hassan Sharghi · Kamran Sartipi

Received: date / Accepted: date

Abstract In-depth analysis of user interactions with applications in large systems is widely adopted as a means to understand user's behavior for strategic purposes such as fraud detection, system security, weblog analysis, social networking, and customer relationship management. Overall, the user behavior presents characteristics, relationships, structures, and effects of a sequence of actions in a specific application domain. The interaction of users with applications at the business-level generates events that make the elements of the user behavior. Formal modelling and representation of complex patterns of user actions using expressive languages are critical aspects of behavior analysis. We present a model to describe the behavior elements and their relationships. The model also provides a systematic mechanism for describing and presenting events, sequence of events, and complex behavior patterns. A behavior pattern can be defined as a sequence of typed events that occur during specific time intervals. An event consists of a tuple of attributes whose values represent an observation of the behavior. In this paper, first we define a semantic model of the user behavior to address the issues around the user behavior representation, and then we present syntax and semantics of a generic Behavior Pattern Language (BPL), which enables the analysts to define a variety of complex behavior patterns in a declarative manner. We present the feasibility of the approach through several examples of complex behavior patterns expressed using the proposed language.

Keywords Behavior Pattern · Sequence of Events · Language · Semantics · Constraint

H.Sharghi
Department of Electrical, Computer and Software Engineering, University of Ontario Institute of Technology, 2000 Simcoe Street North, Oshawa, ON, L1H 7K4, Canada
Tel.: +1-905-721-8668
E-mail: Mohammadhassan.Sharghigoorabi@uoit.ca

K. Sartipi
Information Systems, DeGroote School of Business, McMaster University, 1280 Main Street West, Hamilton, ON, L8S 4M4, Canada
Tel.: +1-905-525-9140
E-mail: sartipi@mcmaster.ca

1 Introduction

Behavior computing is an emerging field and young discipline aiming at exploring formal methods for human behavior representation and analysis. However, representation as a key component in behavior computing has not been sufficiently studied to establish a generic and comprehensive approach [11].

Designing an efficient and scalable infrastructure for monitoring and processing behavior patterns has been a major research interest in recent years to provide a mechanism for the enterprise to monitor the behavior patterns and anomalous behaviors of their customers. Behavior data are becoming a valuable asset to be carefully analyzed in order to reveal its explicit and implicit knowledge that cannot be attained just through recorded transactional data. Moreover, appropriate presentation of behavior analysis results is valuable for end users to enable them to make decisions properly.

Human and machine behavioral modeling and analysis are becoming interesting areas of study in a variety of research domains such as computer security and access control [31, 33], insider threat detection [27], fraud detection [24], finance [10, 17], social network analysis [2], and weblog analysis [22]. However, modeling, representing, comparing, and analyzing of users's behavior have not been explored in a precise way.

Behavior modeling and representation attempt to develop languages and tools based on formal methods and emerging technologies. Such a language will allow the analyst to illustrate primitive events and their attributes, semantics, constraints, and behavior patterns in a detailed and precise manner. The correlations between events based on attribute values along with constraints constitute the semantics of behavior pattern. In this paper, we describe the semantics of behavior pattern in terms of proximity, association, separation and causality relations between events and illustrate them by a series of tangible examples in healthcare domain.

In this context, the proximity pattern refers to a sequence of events that occur within a particular period of time or in a specific location. This is achieved by restricting the attribute values of time or location. Association pattern refers to the sharing of the same group of attribute values among a subset of events, and represents the correlation between events. Separation is another category of behavior pattern that corresponds to two or more associated events that are separated based on the value of a specific attribute value. In causality relation between events, one event can affect another event based on the result of an attribute value evaluation. In this case, the first event or group of events causes the second event or group of events.

Due to the lack of access to the real log data for behavior analysis in critical domains, algorithm evaluation is a challenge for analysts. Consequently, generating log data containing different categories of patterns can relieve the issue of algorithm evaluation. However, some issues exist around generating behavior patterns such as: how analyst can present the structure of the behavior pattern? How the semantic of the behavior pattern can be represented? And how the constraints for the behavior pattern can be demonstrated?

In this paper, we propose Behavior Pattern Language (BPL) that allows the analyst to specify precisely the features of the user behavior patterns. BPL's features have been inspired from high-level programming languages such as Ruby [28, 15] and Python [23]. The defined operators represent relationships between the behavior elements. Negation or non-occurrence, event sequence, frequency of occurrence, and time and location separations between the events, are some important features of the BPL that provide semantics for the behavior patterns.

This paper presents the following contributions: i) provisioning a model based-on ordered set of events to present the feature of the user behavior pattern; and ii) proposing a specification language that offers powerful features to demonstrate the structure and semantics of the user behavior patterns. The remaining of this paper is organized as follows: Section 2 investigates the related work. We describe the proposed framework and our approach to model the behavior pattern in Section 3 and Section 4, respectively. In Section 5 we present the syntax, semantics and feature of the proposed language. In Section 6, different categories of behavior pattern will be modeled by BPL. Finally, Section 7 provides the concluding remarks.

2 Related work

Behavior modeling has been increasingly recognized as a challenge for associating semantics with the human's actions to be used in different environments and for different purposes. Cao et. al [10, 29] consider behavior as an individual's activities represented by events as well as activity sequences issued by the user within certain context. They defined four dimensions, as: actor, action, environment and relationship to represent abstractly the user behavior. The assigned attributes allow for describing features of each dimension and Temporal Logic is used to express the properties and relations between elements of a desired behavior. Their approach is similar to the BDI (Belief-Desire-Intention) model [30] developed in multi-agent systems, and the proposed model is appropriate for modeling and analyzing activities in abstract level.

In [32] a conceptual language NKRL (Narrative Knowledge Representation Language) provides an ontological paradigm to deal with the most common types of human behaviors. This approach is basically a knowledge representation language to fill the gap between behavior modeling and its translation into computer-usable tools. They interpreted a narrative including a sequence of logically structured elementary events that describe the behavior. In [2] the authors introduce a semantic model for representing and computing behavior in online communities. An ontology was defined that represents all involved entities and their interactions. Representing the behavior pattern by semantic rules has been also proposed in literature so that in [9] Event-Condition-Action (ECA) rules was considered to represent the frequent behavior patterns.

The mentioned ontology-based languages for behavior tries to infer the semantics of a behavior described by a natural language. BPL differs from such languages in terms of objective. BPL represents a behavior based on attributes of actual events occurred in the environment. The correlation between events is considered as the semantics of the behavior. BPL leverages the constructs in high level programming languages to represent such correlations instead of using semantic rules whose may convey some kinds of ambiguities.

Representing behavior using the modelling languages has been proposed in [26, 16]. The HBML (Human Behavioral Modeling Language) was introduced by Sandell et. al [26] in order to efficiently capture the relationships and behaviors derived from analysis of data streams. The behavior is represented in three orders (0th order, 1st order and 2nd order) so that the 0th order describes the list of activities that a user may engage to do, the first order describes the 0th order activities along with some environmental context information, and the second order describes the activities with probabilistic characterization.

HBML is a domain specific modelling language for instantiating process models for profiling and tracking the behavior of objects. A profile contains information such as descriptions, attributes, activities and distributions relevant to a specific behavior. HBML defines a behavior as a set of activities with a statistical characterization of those activities. Using the statistical model for behavior representation is completely different from the model applied in the BPL. The HBML uses a probability distribution to describe a new behavior of the entity based on the past behavior. However, modelling a new behavior resulted from the past behavior of the user is not the goal of the BPL for a behavior representation.

Behavior representation and analysis is closely related to the event processing approaches that have been explored extensively in recent years. Complex Event Processing (CEP) [19] is a technique of high speed processing events on a lower abstraction level in order to recognize significant events or meaningful patterns in event stream. A series of SQL based languages have been proposed for querying the desired patterns on event streams [13, 8, 25, 6, 5]. These domain specific languages are useful for making query on event stream, however there is no information about their success in representing the correlation among events and the semantics of the patterns. Languages that have been recently proposed to represent events and their correlations, can be divided into two groups: Data Stream Processing (DSP) and Complex Event Processing (CEP) languages [12].

CQL (Continuous Query Language) is a notable representative of DSP-based languages which is an expressive SQL-based declarative language for registering continuous queries against streams and stored relations [4]. Each query in CQL defines rules on one or more streams as inputs and generates one output stream. The expressiveness of CQL is provided by three classes of operators that are used to define queries. "Stream-to-relation" operators are known as "windows" and are applied to produce a relation from a stream, "relation-to-relation" operators that produce a relation from other relations and are basically standard SQL operators, and "relation-to-stream" operators that produce a stream from a relation. However, no explicit sequencing operators are provided to represent the ordering between elements during handling the stream and relation in CQL and similar languages. Therefore, representing behavior patterns containing sequence of events by CQL is very difficult. In general, DSP-based languages have been designed to separate portions of input streams and perform traditional database operations. Consequently, they have limitation to detect complex patterns.

In contrast, CEP-based languages have been specifically designed for detection of complex patterns on incoming streams. The majority of existing languages support non-monotonic features such as negation of events, aggregation, or repetitive events. However, the lack of expressiveness to represent complex features is a significant challenge for existing languages. For example, [7] introduces a language that implements a data model consists of temporally ordered sequences of event streams so that each event stream has a fixed relational schema. The language in [8] also focused on temporal relation between events by looking at the occurrence

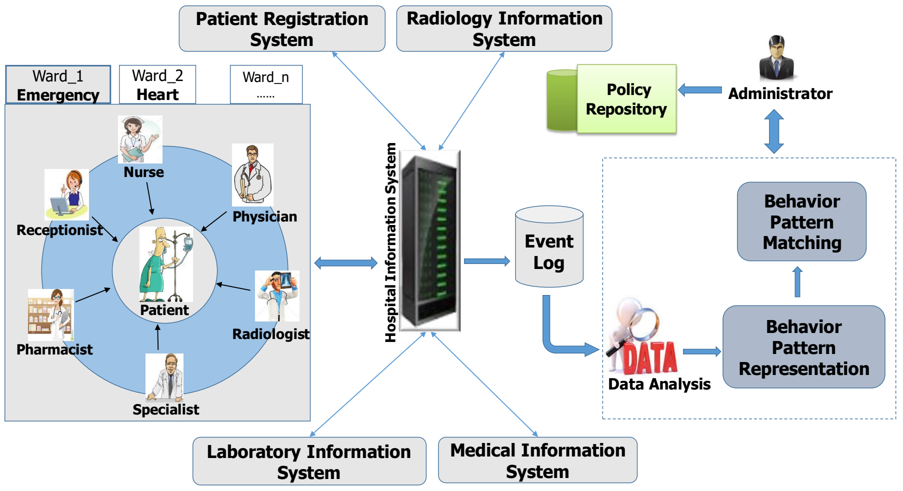

Patient Registration System
Radiology Information System
Ward_1 Emergency
Ward_2 Heart
Ward_n ……
Nurse
Physician
Receptionist
Patient
Radiologist
Pharmacist
Specialist
Hospital Information System
Event Log
Policy Repository
Administrator
Behavior Pattern Matching
Behavior Pattern Representation
DATA
Data Analysis
Laboratory Information System
Medical Information System

Fig. 1: The proposed framework for extracting knowledge about the behavior of authorized users through mining and analyzing event sequences in the system's event log.

times of events. This language uses reactive and deductive rules to reason about relationships of events.

Compared to such languages, the BPL provides any kinds of relationship between events based on attribute values. Therefore, BPL is not limited to only temporal relation. Actually, temporal relation is implicitly considered by using order-set in our proposed language. Meanwhile, the BPL is not just a query-driven language that processes repetitively a query on a stream at given intervals.

Rules, that describe a pattern, are often expressed in natural language that is inherently ambiguous. Therefore, any designed language should have precise semantics for developed features in order to use such rules for event selection, combination and processing [3]. Providing such unambiguous semantics is another challenge for existing languages. Some languages such as [7, 18, 20] attempted to solve the issue through providing strict policies for selection and combination of events, but they do not allow users to change or adjust policies.

To rectify the deficiencies found in existing languages and provide a high degree of expressiveness, we introduce a generic language called Behavior Pattern Language (BPL) to represent accurately a complex behavior pattern. A powerful behavior pattern language is more than just a means for representing a behavior pattern. It also serves as a framework which helps analysts organize their ideas about a behavior or compose behavior pattern from a hypothesis in order to approve or reject a suspicious behavior. BPL provides some mechanisms to combine simple expressions and primitive events to represent the structure and semantic aspect of a user behavior pattern.

3 Proposed framework

Due to the complex structure of a healthcare environment, implementing a secure and reliable information system is a challenging task. As an example, we consider a Hospital Information System (HIS) whose main task is provisioning timely and securely access to different resources for authorized health professionals. To appreciate the complexity of an HIS implementation, one can imagine a large number of interconnected and communicating subsystems that constitute the HIS, where each subsystem is a potential target for abnormal use by its authorized users.

Figure 1 illustrates the proposed framework for extracting knowledge about the behavior of the authorized users through mining and analyzing event sequences in the system's event log. Hospital departments are organized into specialized wards. Experts in different roles work together in teams to treat patients. They communicate with a HIS to acquire data from different systems such as: laboratory information system, medical information system, diagnostic imaging system, radiology information system, etc. Such interactions should follow the job workflow and satisfy the policies defined in the system. However, the IBM Cyber Security Intelligence Index reports that 55 percent of attackers are insiders [1]. The increase in the number of successful insider attacks on healthcare systems in recent years shows that the current access control mechanisms are inefficient in protecting against insider threat.

The behavior of users in different roles with the HIS system to access different kinds of resources will be recorded in a repository, namely event-log. Events convey important information about the behavior of users, and in particular sensitive aspects such as suspicious activities that may jeopardize the system's security. However, it is almost impossible to extract useful knowledge from individual events and attributes, unless we consider the relations such as sequence, association and dependencies among events.

Authenticated and authorized users can affect the security level of a service-based distributed system through misusing the resources. Therefore, the behavior of users should be constantly monitored and the policy of the system should be regularly updated by the administrators. However, monitoring and analyzing the activities of users are not easy tasks, particularly in distributed systems that incorporate a large number of users with different levels of authorization. Moreover, the changing nature of users' behavior poses significant challenges with respect to system monitoring, auditing, and diagnosing.

Supervised analysis of event data to reveal suspicious activities and extract user behavior patterns can assist the administrators to enhance the security level of the system. However, extracting the behavior patterns for the purpose of security is challenging in terms of modeling and representing behavior patterns, and extracting behavior instances from the log data, which requires a pattern-matching engine to identify user-behavior instances in a large and dynamic event log, and perform post analysis of the extracted instances.

As can be seen in Figure 1, all access to resources will be recorded in a central log repository. Statistical analysis of event log provides primitive knowledge about the usage percentage of different resources, usage frequency during different time interval, the spent time for each resource, the frequency of a particular activity on a resource, user's demography, etc. Log data analysis can supplement the understanding of users' behavior with more concrete data, and it provides a means to investigate the users' interest to existing resources in a distributed system. The statistical information provides some hints for data analysts to explore about particular behaviors in the system. Modelling and representing such behaviors are the significant challenges in this context, and the main motivation for this research

Table 1: Sample attribute names and domain values

Attributes and Domain Values	
Attribute Name: a	**Domain of Values:** $\mathrm{D_a}$
User	John, Mike, Emma, Julia, Mary, Robert
Role	Nurse, Physician, Radiologist
Action	Read, Write, Create, Order, View
Resource	Diagnostic Report, Image, Exam
Time	00:00 − 24:00
Date	2000-01-01 , 2099-12-30

is to propose a state-of-the-art solution through proposing a Behavior Pattern Language (BPL).

Data analysts can utilize the proposed BPL to describe any suspicious behavior for investigation. The described behavior will be converted into a sequence of events called reference pattern and a set of constraints that represent the correlation between events. Next, a pattern matching engine searches the log file to identify patterns that approximately match the reference pattern.

We used the Valued Constraints Satisfaction Problem (VCSP) to model the approximate behavior pattern matching problem. A VCSP is characterized by a set of hard constraints that must be satisfied, and a set of soft constraints whose satisfaction is desirable. Therefore, solving a VCSP means finding an assignment (set of events) that sub-optimally satisfies a set of hard and soft constraints. In other words, the VCSP searches event logs to identify instances that are highly similar to the reference pattern. Analyzing the results reveals some facts about users, resources and actions. The administrator can use such facts to react properly against abnormal behavior or enhance system security by the means of updating the access control policies of the subsystems.

4 Behavior modeling

In this section, we formally define different components that progressively build the proposed user behavior model.

Definition (Attribute a): an attribute "a" ($a \in A$ set of attributes) is a contextual information from an actionable and context-aware environment such as hospital, business enterprise, or any other work place. The value of attribute "a" belongs to a particular domain of values $\mathrm{D_a}$. Typical primitive types such as integer, float, text, date and time are used to define the domain values of an attribute. Table 1 specifies the attribute names and their domain values in the context of this paper. *Definition (Event e):* an event is a primitive element of a behavior. Events represent interactions between behavior subjects and objects within the system. An event is represented as a tuple of attributes that describe the characteristics of the interaction between a user and a system resource. An analyst can obtain some basic information from a primitive event since the semantic information provided by a single event is quite limited. Below, two examples of events with six attributes are shown:
e1= <John, Nurse, Read, Diagnostic Report, 10:00, 2015-06-10>
e2= <John, Nurse, View, Image, 10:30, 2015-06-10>

Definition (Event set E): an event set is the set of all primitive events that are recorded in the log repository of the system. Basically, the event type defines a specification for an event set so that all events should have the same structure.

Definition (Event type T): an event type is determined by the structure of the event tuple in terms of the number of attributes and the domain values of attributes. Two events have the same type if both the number of attributes and their domain values are the same (as e1 and e2 above). Below the event type associated with events e1 and e2 are shown:
T = <User, Role, Action, Resource, Time, Date>

Definition (Behavior B): a behavior is an ordered-set[1] of finite number of events that possess the same event type T. For simplicity we also refer to the ordered-set as a sequence. The order of events in the set is determined by an index, which starts with 1. Each event in the set is denoted by a single variable e_i. The occurrence order of the events is based on a time stamp. For example, e_{i-1} is before e_i and e_{i+1} is after e_i, however, the time intervals between the events may be different.

$B= \{ e_1, e_2, ..., e_n \}$

Definition (Behavior pattern P): a behavior pattern consists of two parts "static" and "dynamic". The static part defines the structure of the pattern and includes the number of events and the event type. In fact, it represents the common characteristics (or context) among the events. The dynamic part defines the semantic of the behavior through making correlation between the events in the pattern. Furthermore, a behavior pattern includes a set of constraints that must be satisfied by the user behaviors that approximately match the behavior pattern. These constraints manage the sequence and association among the events. In other words, the semantics of the behavior pattern is represented by a set of constraints that provide association among the events or specify particular conditions for the pattern to be met by the matching user behaviors.

Definition (Feature f): a feature "f" ($f \in F$ set of features) of the behavior pattern is a relation between events based on attribute values. It is defined by a tuple containing the following elements:

f = <E1, E2, T, V, O> where:

E1, E2 $\subseteq$B are ordered sets of events

T: Event type

V: Ordered-set of attribute-domains (different D_a's), where the order of attribute-domains in V is the same as the order of attributes in T.

O: Set of operations (defined below)

Definition (Operation o): an operation "o" ($o \in O$, where O $\subseteq E_1 \times E_2 \times T \times V$) defines a relation between two sets of events, attributes and values. An operation synthesizes a particular feature by defining constraints between event-sets.

Definition (Constraint C): a constraint C_j is defined as a pair $<t_j, R_j>$where $t_j \subseteq A$ is a subset of k attributes (defined by event type T) and R_j is a k-ary relation on a set of attribute-domain subsets D_j for the corresponding attributes t_j. In this case, an evaluation of the attributes (i.e., assigning values to attributes) assigns values from the subsets of domains (D_j) to a particular set of attributes, such that the relation R_j is satisfied.

[1] Behavior is an ordered-set of events since every event in the behavior is unique due to its time occurrence.

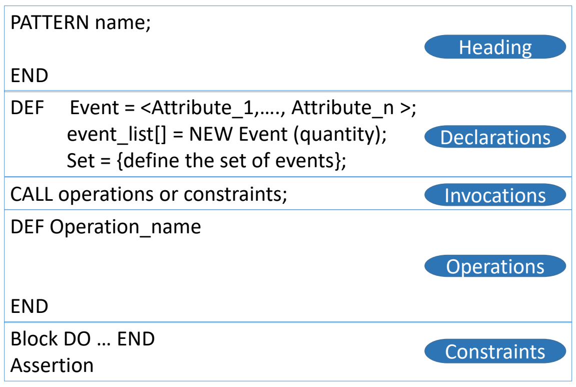
5 Behavior pattern language

Our definition for behavior which is based on a finite ordered-set of events allows our behavior language to represent the user behavior through a set of operating atomic events. Behavior representation and analysis are close to Complex Event Processing (CEP) as an emerging field that attempts to detect event patterns using continuously incoming events based on an abstraction level. Using general-purpose languages such as C or Java for implementation of event processing features causes extra and complex operations in terms of implementing low level functions for events and query structures. Therefore, designing a specific language for event processing has been emerged as an interesting topic for research. Several languages have been introduced as the result of such research in recent years.

To justify why we need a new event-based specification language, we present an example to illustrate the kind of primitive expressiveness and flexibility we need to handle users' behavior patterns. As an example, we consider a hospital information system (illustrated in Figure 1) comprising of several systems in order to provide facilities for authorized care providers (users) to access resources such as medical information, laboratory information, diagnostic images and associated reports. Events representing users' interactions with the resources will be logged in the system's storage. Each event is described by a set of attributes. Now, suppose the administrator must be notified in case of misusing the resources. Depending on the context, the application requirements, and the user preferences, the notion of resource misuse can be defined in different ways. Here we present four cases of misuse behavior and use them to show specifications that should be provided by an event processing language. Misuse can occur when:

i. A nurse first reads and then writes a diagnostic report pertinent to a specific patient more than 3 times within 5 minutes.
ii. A user edits the diagnostic reports of a patient in ward-1 in the morning, and again she accesses those resources when she works in ward-2 in the afternoon.
iii. A physician accesses to resources more than twice the number of her assigned patients within a month.
iv. A user sends requests to access the system resources in less than 3 minutes from two different locations whose distance is more than 100 meters.

Such cases apply a set of constraints to select the relevant events from the dataset. Two kinds of selection constraints have been used. The first kind chooses events based on attribute values and we call it "*attribute parameterization constraint*". For example, case (ii) considers those events that occur in particular locations, or case (iii) addresses events whose role attribute value is "physician". The second kind of selection constraints operates on a particular relationship among events, namely "*attribute factorization constraint*". For instance, case (i) affects those events that occur within 5 minutes, actually it defines a timing relationship between events in order to select a sequence of events, or case (iv) filters events according to a location proximity. Furthermore, case (i) combines both parameterization and factorization constraints to define iteration that selects those sequences of events that repeating specific actions. Case (ii) implies negation through limiting the occurrence of events in a given interval. Similarly, case (iii) introduces aggregation by applying arithmetic functions on events relevant to a particular role



```
PATTERN name;

END
DEF    Event = <Attribute_1,...., Attribute_n >;
       event_list[] = NEW Event (quantity);
       Set = {define the set of events};
CALL operations or constraints;
DEF Operation_name

END
Block DO ... END
Assertion
```

Fig. 2: The structure of a BPL code to describe a typical behavior pattern

in order to compare the rate of access to resources and the number of assigned patients.

Finally, when the events have been selected, such cases specify which pattern will be created by those events. Moreover, the inner structure of the pattern in terms of length, sequence of events, and their association will be defined by such cases. A language for behavior pattern not only should be simple and unambiguous for translating constraints, but it also should be able to express all the defined constructs: parameterization, factorization, iteration, negation, aggregation, and sequencing.

BPL is a language to represent the user behavior through defining a sequence of events and their correlations. The ability to combine events and keep the ordering of events is a basic and powerful specification for an event-based language. Consequently, we deal with two kinds of elements in BPL: ordered sets of events and operations. Operations are descriptions of the constraints for defining the correlations between events. The embedded features and operators to BPL improve the capability of BPL to address any categories of behavior.

5.1 Syntax and grammar of BPL

The BPL code consists of five parts to represent a typical behavior pattern. Figure 2 illustrates the structure of a BPL code specification. The declaration is used to describe different entities existing in the pattern. These entities include: attributes, events, sequence of events and some parameters that specify the structure of the pattern. In other words, the skeleton of pattern will be defined in the declaration section, then the pertinent operations and constraints will be called to establish the meaning of the pattern.

The methods in the operation section are used to define the relation between two ordered sets of events that are chosen by the analyst. These ordered sets are subsets of the sequence of events that represent the pattern. The BPL allows the analyst to choose at most two sets because the defined operators are unary or binary. The relation can be defined through applying different statements and operators to manipulate attributes and attribute values. If the analyst intends to define particular restrictions for the operation, the relevant assertions will be defined in the constraint section. The methods in the operation section interact with the constraints via the provided language constructs such as block and yield.

We used the Extended Backus-Naur Form (EBNF) to describe the syntactic structure of the BPL. Appendix A provides the syntax of the BPL.

5.2 Features of BPL language

BPL takes advantage of the following concepts in high-level programming languages, to define different forms of behavior patterns.

Iterator: Most advanced programming languages provide an object called iterator that enables a programmer to traverse a collection of data. Each data item is considered once during a traversal. The iterator handles the progress of the iteration by providing a reference to the next entry in the collection. We provide this capability for BPL through "each" method and "foreach" statement that can manipulate repeatedly the events in an ordered set.

Block: The structure of block is extensively used to break up large programs into tractable pieces. It is an important tool for organizing the construction of large programs. Blocks can appear immediately after the invocation of a method. Similar to a method, the block can take parameters. We consider that the statements of the block are located between "do - end" keywords, and the parameters appear at the beginning of the block and between the vertical bars. In the following example, we assume that B is an event-set containing 5 events of type T that uses an iterator (each method) and a block to assign values to attribute Role for all events. Each event in *B* is fetched by iterator *each* and then passed as an argument that is replaced with parameter *event*. The block has one statement that assigns value "Nurse" to each fetched event.

```
-- Assigns values to attribute Role

B = {e1, e2, e3, e4, e5};
B.each do |event|
    event.Role = "Nurse";
end
```

Module: A module is a programming concept which allows for reusability of code through grouping together methods, classes, and constants. To facilitate the representation of different kinds of patterns, we define a series of methods called operators to manipulate events. These built-in operators can be organized as modules and then can be utilized to represent different categories of behavior patterns.

Yield: A common use of yield in a programming language is to transfer the control from one point of the code to another point of the code. This capability enables programmers to make a customizable form of an iteration. However, in some situations, it needs to implement or reuse a custom functionality inside a common functionality. We leverage the feature of yield construct in Ruby to implement constraints as custom functionality inside operations as common functionality. In other words, we use yield to separate the parts of the representation that demonstrate the semantics of the pattern from those that define the structure of the pattern.

5.3 Applying the features

In Example 1 below, we apply the mentioned features in order to define an operation whose task is to assign values to two attributes of every event in the ordered-set B consisting of 5 events. When operation "OP" is called via "CALL OP (B)", the ordered-set B is passed to it as parameter. In definition of "OP" the keyword "each" plays the role of an iterator and fetches individual events from the ordered-set B repeatedly and passes them to a block as parameters. The block contains a "YIELD" statement which passes each event "e" back to another block that is defined after "CALL OP (B)". This second block then assigns the attributes "User" and "Role" of each event to "John" and "physician". After executing the statements of the second block, the control returns to the "OP" and this procedure repeats for the next event.

```
--Example 1;
B = {e1, e2, e3, e4, e5};
  CALL OP (B) DO |event|
      event.User ="John"
      event.Role ="physician"
   END

   DEF OP (set)
       set.each DO |e|
           YIELD (e)
        END
   END
```

Below is another example that demonstrates the functionality of yield so that the common functionality is handled by "method2" and custom functionality is handled by the block defined in "method1".

```
DEF method1
    CALL  method2   DO  |b|
               FOREACH event IN b  DO
                   event.Action = "write";
               END
    END  // end of block
END  // end of method1

DEF  method2
        B = {e1, e2, e3};
        YIELD (B)
END
```

When method1 is invoked, it calls method2. method2 starts executing until yield is reached. At that point control along with the parameter B is transferred to the block that was defined in method1. The block is executed. The value of attribute "Action" changes to "write" for all events. Then, the control returns to method2.

5.4 Operation and Operator

What makes a user behavior pattern distinct from another pattern is the properties that exist in the sequence of events representing the user behavior pattern. Correlations between events convey the meaningful features of the pattern. The analyst can show the features of the pattern through defining particular operations on the ordered set of events. Assigning events to event-sets depends on what events participate in the association and how many event-sets are needed. We consider at most two ordered sets can be operated in each operation because unary and binary operators will be used in an operation. To represent a feature, the analyst is allowed to adjust the sets in order to define appropriate operations. To implement the relation among events inside an event-set or between two event-sets, we define several unary and binary operators. The defined operators can be grouped in three categories.

- The first category containing "NEW" and "INITIALIZE" is intended to define the structure of the pattern. The operator "NEW" creates sufficient placeholders for corresponding events in the pattern. Events in a pattern have the same type, so the placeholders have the same size and structure. The operator "INITIALIZE" assigns the value to a particular attribute for all events in an event-set.
- The second category of operators are used to manipulate the ordered set of events. The operator "VALUEOF" returns the value of an attribute for a particular event or for all events. Joining two event-sets can be performed by operator "UNION" so that the result of union is a new ordered set of events. The operator "FOLLOWEDBY" is useful to change the sequence of events in an event-set. The specification of a pattern can be expressed by constraints. Not only do constraints define the correlation between events, but some kind of constraints also affect the whole pattern. The appropriate operations manage the constraints that make correlation between events. However, to handle the second kind of constraints that limit some features of the whole pattern, we use the built-in operators or the combination of operations and operators. In BPL, we also utilize the assertion mechanism (by using keyword "ASSERT") to enforce these kinds of constraints.
- The third category of operators include "DISTANCE" and "FREQUENCY" that are designed to apply constraints. Calculating the difference of two attribute values can be used to define particular attribute based constraints. We define the operator "DISTANCE" in order to calculate the distance between values of attributes whose types are time or location. It computes the distance of attribute values for each pair of event. It returns an array as the result of difference. For example, we want to enforce three events e1, e2, e3 occur within 2 hours. The DISTANCE (e1, e2, e3, Time) returns the following array that the value of each element should be less than 2.
  $[\,(e1.Time - e2.Time), (e1.Time - e3.Time), (e2.Time - e3.Time)]\ \leq 2$
  The operator "FREQUENCY" enforces the number of times a set of events (representing a sequence of actions) should be repeated in the dataset.

5.5 Semantics of BPL

Semantics provides the meaning of the well-formed statements of a language. Semantics should be preserved by language implementation applications such as interpreter or compiler. The syntax is specified by a context-free grammar such as Backus Naur Form (BNF) and its variations. However, there is not a single definition method to explain the semantics. An informal way (i.e. example or text) is usually used to specify the semantics of the language constructs [14].

Operational semantics presents the meaning of a program in terms of how it can be interpreted by a machine. In other words, the operational semantics

Table 2: The description and syntax of operators

Operators	
Operator	**Description**
NEW	*NEW event-type (number of event)*, returns an event-set that have same type
INITIALIZE	*INITIALIZE (ordered set of events, attribute, set of value)*, initialize an attribute for all events in the set. The cardinality of event-set and value set is the same.
VALUEOF	*VALUEOF (ordered set of events or an event, attribute)*, return the value of attribute for each event in the set or for a particular event.
UNION	*UNION (ordered set of events, ordered set of events)*, make the union of two ordered set. The result is an ordered set as well.
FOLLOWEDBY	*FOLLOWEDBY (event, event)*, the first event is followed by the second event.
DISTANCE	*DISTANCE (ordered set of events, attribute)*, enforces the distance between attribute values of all generated pairs of events. It can be used for attributes whose type is time or location. The return result is an array containing the difference between attribute values of pairs of events.
FREQUENCY	*FREQUENCY (ordered set of events)*, enforces the frequency of the ordered set in the dataset.

provides a translation mechanism to convert a program to an equivalent program in another language which is simple for understanding. Plotkin [21] introduced the operational semantics in order to describe language semantics in terms of a state transition system as a mathematical tool. Such view makes the operational semantics similar to an interpreter.

The operational semantics is defined by state transitions of an abstract machine. An abstract machine consists of four main elements: control stack (c), result stack (r), processor and memory. Instructions are stored in the control stack and the intermediate results are kept in the result stack. A processor performs operations such as comparisons, arithmetic and Boolean. Variables and values are stored in a memory modelled by a function called m. $dom(m)$ denotes the set of locations where m is defined. $m(l)$ denotes the value stored at location l and $m[l \mapsto n]$ maps l to the value n. Formally, an abstract machine is defined by a set of states (configurations) along with transition rules. A state s is defined by a triple $\langle c, r, m \rangle$ of control stack, result stack, and memory. A transition rule is a binary relation between two states.

Definition 1 (semantics of expressions): The value generated by the machine after zero or more transitions for an expression E is considered as the semantics of E [14].

$\langle E.c, r, m \rangle \xrightarrow{*} \langle c, v.r, m' \rangle$
$E.c$: denotes the expression of E on top of the control stack
$v.r$: denotes the value of E on top of the result stack
$\xrightarrow{*}$: denotes zero or more transition steps

Representing the semantics of a language through using abstract machine needs to write a large number of transition rules in detail. Although it is useful for the implementation of the language, it is too detailed for the users of the language. Therefore, we applied structural operational semantics proposed in [21] based on the transition system. In structural operational semantics, the transition rules for a compound statements are inductively defined based on transition rules of substatements.

There are two methods to represent the structural operational semantics [14]: (1) small-step semantics, based on a reduction relation, and (2) big-step semantics, based on an evaluation relation. In the first method, a transition relation is inductively defined by a set of axioms and rules between states. Axioms are used for reduction whereas rules are applied to specify how to generate new reduction steps based on ones that have been already defined.

The big-step semantics associates each state with its result and abstracts from the details of the evaluation process. The transition systems have the same set of states in big-step semantics and small-step semantics, but the transition rules are different. Transition rules represent individual computation steps in the small-step semantics and full evaluation in the big-step semantics. We use the big-step operational semantics to explain precisely the features of BPL. The transition relation is denoted by $\Downarrow$.

Variable and Constant: We define Rule 1 and 2 to describe the meaning of constant and variable respectively.

$$\langle c, s\rangle \Downarrow \langle c, s\rangle; if\ c \in \mathrm{D_a}\ \cup\ \{True, False\} \tag{1}$$

$\mathrm{D_a}$ denotes domain value for an attribute a. c represents the constant. s shows the state of the transition system.

$$\langle *l, s\rangle \Downarrow \langle v, s\rangle; if\ s(l) = v \tag{2}$$

l is a location of memory (variable) and $*l$ denotes the content of the location l and $s(l)$ represents the value of l in the state s.

Operator: The meaning of a statement that uses a binary operator such as arithmetic or comparable is defined by Rule 3. The rule says that for an expression (e.g $E1\ op\ E2$) containing a binary operator, first the value of E1 and then the value of E2 are computed. The final result is generated by applying the operator on the both values.

$$\frac{\langle E1, s\rangle \Downarrow \langle n1, s'\rangle \quad \langle E2, s'\rangle \Downarrow \langle n2, s''\rangle}{\langle E1\ op\ E2, s\rangle \Downarrow \langle n, s''\rangle\ if\ n = n1\ op\ n2} \tag{3}$$

Iteration: Iterative constructs cause a statement or sequence of statements (the body of the loop) to be repeated for each item in a collection. BPL uses *foreach* and *each* for iteration whose abstract syntax is: *foreach A do B*; where A is a set whose elements will be fetched one by one and B is the body that should be executed for each element. Rule 4 shows the semantics of the iteration. The rule indicates that when an element is successfully returned, the state of program will be changed to s' and the returned element will be saved in location l. The statements in B can use the stored element in the state s' and the execution of statements changes the state to s''. The *skip* means the control is successfully passed to the next statement. Finally, the iteration started in the state s terminates successfully producing s'''.

$$\frac{\langle A, s\rangle \Downarrow \langle true, s'[l \mapsto v]\rangle \quad \langle B, s'\rangle \Downarrow \langle skip, s''\rangle \quad \langle foreach\ A\ do\ B, s''\rangle \Downarrow \langle skip, s'''\rangle}{\langle foreach\ A\ do\ B, s\rangle \Downarrow \langle skip, s'''\rangle} \tag{4}$$

Operation: BPL calls operations to represent relationships between the behavior elements (events). Each operation has a name, definition and may receive several parameters. We represent abstractly an operation in BPL as an expression of the form $f(x_1, \ldots, x_k) = d$ where $x_1, \ldots, x_k$ denote the parameters and d represents the body of the operation f. Rule 5 indicates how an operation is interpreted. First, the arguments of the operation are evaluated and then values are used in the definition of the operation. In other words, the parameters in the body of the operation are replaced with the values of the arguments. The *skip* in the rule shows the control returns to the next statement after calling the operation. We assume that arguments are sent by the reference. Consequently, the results of the operation are directly assigned to the address (l_i) of arguments.

$$\frac{\langle x_1, s\rangle \Downarrow \langle v_1, s'\rangle \ldots \langle x_k, s\rangle \Downarrow \langle v_k, s'\rangle \quad \langle d[x_i \mapsto v_i], s'\rangle \Downarrow \langle skip, s''[l_i \mapsto v_i']\rangle}{\langle f(x_1, \ldots, x_k), s\rangle \Downarrow \langle skip, s''\rangle} \tag{5}$$

Block: Unlike the operation, the block does not possess a name however it can accept parameters. We consider the block as a sequence of statements between keywords *do* and *end*. We define two rules to give a precise meaning to the construct of the block. Rule 6 shows the assignment of the actual value (argument) v to the parameter p. The assignment will be done by a reference (pointer) so that the value of *ref p* is a new location whose content is the value of p.

$$\frac{\langle ref\ p, s\rangle \Downarrow \langle l, s'[l \mapsto v]\rangle}{\langle p, s\rangle \Downarrow \langle v, s'\rangle} \tag{6}$$

Rule 7 shows the meaning of the block definition. According to Rule 6, the parameter is referenced to the argument and then the value of argument (content of reference) is assigned to the parameter for each command C. After executing the command, control will be transfered (*skip*) to the next command and the content of the reference will be updated.

$$\frac{\langle p, s\rangle \Downarrow \langle v, s'\rangle \quad \langle C[p \mapsto v], s'\rangle \Downarrow \langle skip, s''[l \mapsto v']\rangle}{\langle DO\ p\ ;\ C\ END, s\rangle \Downarrow \langle skip, s''\rangle} \tag{7}$$

Yield: Using *yield* operator was inspired by Ruby programming language so that such feature allows programmers to write code succinctly and efficiently. An operation can be called with parameters as well as a block that itself may possess parameters. We represent abstractly the block as a tuple $\langle p, b\rangle$. p is the parameter and b is the body of the block. Such a block will be invoked by a yield statement defined inside of the operation definition. The values of block's parameters will be provided by yield statement. We assume that the operation call happens in the state s_1 as well as all parameters and statements of the block will be kept at this state. The operation starts running until it reaches the *yield*. Rule 8 shows the assignment of the actual value v to the parameter a. We consider a as the parameter of the *yield*.

$$\frac{\langle ref\ a, s\rangle \Downarrow \langle l, s'[l \mapsto v]\rangle}{\langle a, s\rangle \Downarrow \langle v, s'\rangle} \tag{8}$$

Rule 9 indicates that the address of the block's parameter will be mapped to the same address containing the value of the parameter a. Then, the control will be jumped to the state s_1 including the body of the block.

$$\frac{\langle a, s\rangle \Downarrow \langle v, s'\rangle \quad \langle ref\ p, s'\rangle \Downarrow \langle l, s_1[l \mapsto v]\rangle}{\langle yield, s\rangle \Downarrow \langle jump, s_1\rangle} \tag{9}$$

The parameter of the block will be replaced with the reference to the actual value and Rule 10 shows after executing the body of the block, the control will be transfered (jump) to the state s' and the content of the reference will be updated.

$$\frac{\langle p, s_1\rangle \Downarrow \langle v, s_1'\rangle \quad \langle b[p \mapsto v], s_1'\rangle \Downarrow \langle jump, s'[l \mapsto v']\rangle}{\langle yield, s\rangle \Downarrow \langle skip, s''\rangle} \tag{10}$$

6 Categories of behavior patterns

Behavioral patterns describe interactions between users and resources in the system. An interaction consists of a set of primitive events that are somehow related to each other. Sequence, association and combination of the events are important relations that can define different categories of behavior patterns in the real world. In addition, there is a special type of relationship called causal relationship that can express a pattern so that the events are not only in association, but that one causes the other. Sequence is handled by temporal attributes of events. Therefore, BPL always uses an ordered set of events to represent a behavior pattern and the index of event indicates the order of occurrence of events. Applying constraints on attribute values usually makes a relationship between events. For example, a proximity constraint on attributes time and location can represent patterns that occur in a particular period of time or within a specific location. The BPL has enough capability to model the proximity of events based on a location or range of time. Example 2 and example 3 demonstrate the capability of BPL to represent the proximity feature of behavior pattern.
Example2:
– Specifies the proximity of events based on a time period,
– Assigns the behavior pattern of a user in any role who first reads and then writes the patients' diagnostic reports more than 10 times from 9am to 10am.

```
PATTERN Example2;
BEGIN
  DEF Event = < User, Role, Action, Resource, Time >;
  E[] = NEW Event (2);
  set1 =  { E[1] };
  set2 = { E[2] };
  attVal = {<Action, ("read","write")>,
         <Resource, ("diagnostic report")>, <Time, (9:00, 10:00)>};

 CALL Op1 (set1, attVal) DO |event|
         event.Action = attVal[Action].value[0];
         event.Resource = attVal[Resource].value[0];
   END

 CALL Op2 (set2, attVal) DO |event|
        event.Action = attVal[Action].value[1];
        event.Resource = attVal[Resource].value[0];
    END

  ASSERT VALUEOF (set1, User) == VALUEOF (set2, User);
  ASSERT FREQUENCY (UNION (set1, set2)) > 10;
```

```
END

 /* define operation and relevant constraint */
 DEF Op1 (set1, attVal)
         set1.EACH DO |e|
         YIELD (e)
    END
    ASSERT Time >attVal[Time].value[0];
    ASSERT Time <attVal[Time].value[1];
 END

 DEF Op2 (set2, attVal)
         set2.EACH DO |e|
         YIELD (e)
    END
    ASSERT Time > attVal[Time].value[0];
    ASSERT Time < attVal[Time].value[1];
 END
```

We define a list called *Event* including the attributes that make the structure of a typical event for the example 2. Then, we use the operator *NEW* to specify that the pattern has two events: $E[1], E[2]$. In this case, each of them is assigned to a set. The BPL uses a dictionary data structure to specify the concrete values for attributes. In this example, the variable *attVal* is a dictionary so that the keys and the list of values have been extracted from the provided scenario. A built in method called *value* is used to access an item of the list in the dictionary. For example, $attVal[Action].value[0]$ means that the first item ("read") in the list corresponding to key "Action" in the dictionary represented by the variable "attVal". We invoke the first operation (Op1) along with a block that has a parameter called *event* by using keyword "CALL". The operation Op1 itself has two parameters: $set1, attVal$. Inside of the operation Op1, the block will be invoked for the event in the set by using operator *yield.* The statements of the block specify some attributes of the event should have concrete values. Moreover, the used assertions emphasize that the value for attribute "Time" should be in the defined range.

Example3:
– Specifies proximity of events based on location and time span.
– Defines the behavior pattern of a physician who can read the patient's diagnostic report from two different wards in a hospital within 2 hours.
–We assume that the distance between two wards is less than 100 meters.

```
 PATTERN Example3;
 BEGIN
         DEF Event = < Role, Action, Resource, Location, Time >
         E[] = new Event (2);
         set1 = { E[1], E[2] };
         attVal = {<Action,("read")>, <Resource, ("diagnostic report
             ")>,<Role,("physician")>, <Location, ("L1","L2")>};

         CALL Op2 (set1, attVal) DO |event|
                 event.Role = attVal[Role].value[0];
                 event.Action =attVal[Action].value[0];
                 event.Location = attVal[Location].value[0] | attVal
                     [Location].value[1];
                 event.Resource=attVal[Resource].value[0];
         END
         ASSERT DISTANCE (set1, Time) < 2;
         ASSERT DISTANCE (set1, Location) < 100;
 END
```

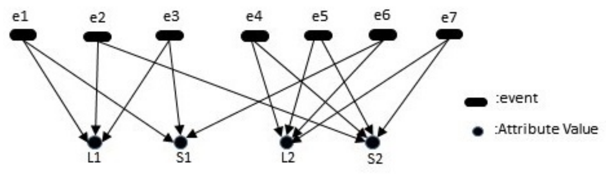


Fig. 3: Association between two separated groups of events

```
 /* define operation */
 DEF Op2 (set1, attVal)
                set1.EACH DO |e|
                YIELD (e)
        END
 END
```

Some behavior patterns include events that can be put into some groups based on a particular context such as location, and time. However, some events share attribute values between groups. As an example, we consider a physician who works in two different clinics (L1 and L2), but he can read the resources from every clinic. A behavior pattern of this user has been shown in Figure 3. Event e2 that is occurred in location L1 can read Resource S2 in location L2. When the user as a physician works in L2 also can read resource S1 located in L1.

Example4:
– Assigns association between two separated groups of events.

```
PATTERN Example4;
BEGIN
        DEF Event = < User, Role, Action, Resource, Location >
        E[] = NEW Event (7);
        set1 = { E[1] ..  E[3] };
        set2 = { E[4] ..  E[7] };
        attVal = {<Action,("read")>, <Resource, ("S1", "S2")>, <
            Location, ("L1","L2")>, <Role, "physician">};
        CALL Op1 ( set1, set2, attVal) DO |event, Lvalue, Rvalue,
            Avalue|
                event.Location = Lvalue;
                event.Role = Rvalue;
                event.Action = Avalue;
        END
        set1 = { E[1], E[3], E[6] };
        set2 = { E[2], E[4], E[5], E[7] };
        CALL Op2 ( set1, set2, attVal) DO |event, value|
                event.Resource = value;
        END
END

/* define operation */
DEF Op1 ( set1, set2, attVal)
        Set1.EACH DO |item1|
                YIELD (item1, attVal[Location].value[0],attVal[Role
                    ].value[0], attVal[Action].value[0] )
        END
        Set2.EACH DO |item2|
                YIELD (item2, attVal[Location].value[1], attVal[Role
                    ].value[0], attVal[Action].value[0])
        END
END
```

```
DEF Op2 ( set1, set2, attVal)
        Set1.EACH DO |item1|
                YIELD (item1, attVal[Resource].value[0])
                END
        Set2.EACH DO |item2|
                YIELD (item2, attVal[Resource].value[1])
        END
END
```

Separation is a category of behavior patterns in which events can be divided in some groups. Meanwhile, the groups affect each other based on particular attribute values. To describe a separation behavior pattern, we consider the workflow that occurs in a clinic to treat a patient. The triage nurse's duties typically consist of measuring the pertinent vital signs and identifying the main complaint. She documents the conditions of the patients and assigns an examination room to the patient. She selects and notifies a specialist based on the patient's main complaint. The specialist is responsible for taking the patient's history, performing an examination, making a diagnosis, and writing a prescription. After all ordered services have been completed; the patient is discharged from the clinic. In the example

Table 3: Events and actions for example 5

EventID	**Action**
e1	A1:measuring the vital signs
e2	A2:recording the main complaint
e3	A3:documenting in EHR
e4	A4:assigning an examination room
e5	A5:notifying the specialist
e6	A6:taking the patient's history
e7	A7:performing an examination
e8	A8:making a diagnosis
e9	A9:writing a prescription

Table 4: Complaint and relevant role for example 5

Complaint	**Role**
C1	R1: Orthopedist
C2	R2: Urologist
	R3: Nurse

illustrated in Figure 4, the actions for treatment of a patient are divided into two groups of sequential events, which are generated by two different roles: nurse and specialist. The attribute value of role in the second group is defined by user who has specific role in the first group, and it is assigned based on the value of the attribute "complaint".

Example5:

– Separation

```
PATTERN Example5;
BEGIN
```

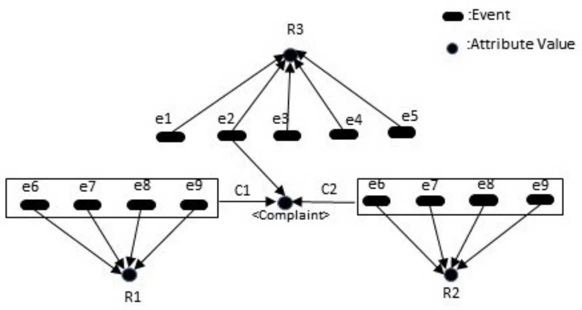


Fig. 4: Grouping of events based on a particular attribute value

```
        DEF Event = < User, Role, Action, Patient, Complaint >
        E[] = new Event (9);
        set1 = { E[1], E[2], E[3], E[4], E[5] };
        set2 = { E[6], E[7], E[8], E[9]};
        attVal = {<Role, ("R1", "R2","R3")> }
        Call Op1 (set1, attVal) DO |event|
                event.Role=attVal [Role].value[0];
        END
        Call Op2 (set1, set2) DO |event, v|
                event.Role=v;
        END
END

/* Define Operation */
DEF Op1 (set, attVal)
        set.EACH DO |e|
                YIELD (e)
        END
END

DEF Op2 (set1, set2)
        Set2.EACH DO |e|
                IF (VALUEOF (set1.event[2], Complaint)) == "C1"
                    THEN
                        YIELD (e, "R1")
                IF (VALUEOF (set1.event[2], Complaint)) == "C2"
                    THEN
                        YIELD (e, "R2")
        END
END
```

Causality represents a case in which event e2 occurs after event e1 not because of the temporal (sequence) relation, but due to the structure of the behavior pattern, and as a result of some attribute value evaluation. If events e1 and e2 satisfy such a condition we say that e1 caused e2 and show this relation by a directed path in the diagram starting at e1 ending at e2. Due to the definition of causality, the first event (cause) should be prior the second event (effect); therefore, an overlap exists between causal and sequence relations, as shown in Figure 5.

Chemotherapy consists of a sequence of events, such as the administration of appropriate dosage of medication and laboratory examinations. The severity of illness diminishes when dosage is increased. However at some point, the patient begins to experience negative side effects associated with excessive dosage. The

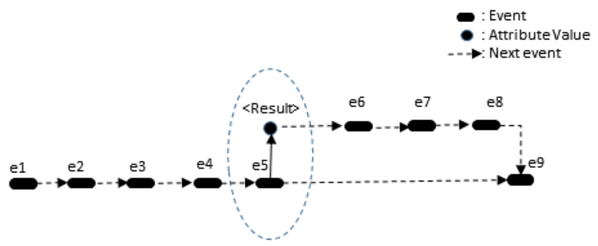


Fig. 5: The causality relation between events in a sequence

oncologist is responsible for monitoring the treatment progress and control the negative side effects through changing the dosage or prescribing other medications.

As an example, we consider the behavior pattern of an oncologist who prescribes particular medication that patients need; also, she monitors regularly the blood-cell count results to make decision about the medication's dosage. If the exam result indicates that the treatment has decreased the number of blood cells below X quantity, the oncologist may decide to stop changing the medication and order new prescription.

Table 5: Events and actions for example 6

EventID	**Action**
e1	A1: Choosing a medicine
e2	A2: Calculating the dosage
e3	A3: Sending the prescription to pharmacy
e4	A4: Ordering an exam
e5	A5: Viewing the blood count result
e6	A6: Changing the medicine
e7	A7: Calculating the dosage
e8	A8: Sending new prescription to pharmacy
e9	A9: Continuing the treatment

Example6:
– Causality relation

```
PATTERN Example6;
BEGIN
        DEF Event = < User, Role, Action, Patient, Result >
        E[] = new Event (9);
        set1 = {E[1] .. E[9]};
        attVal = {<Result, (value)> }
        CALL Op1 (set1, attVal);
END

DEF Op1 (set, attVal)
BEGIN
        IF ( VALUEOF (set.event[5], Result) )  > attVal[Result].
            value  THEN
                FOLLOWEDBY (set.event[5], set.event[9]);
END
```

7 Conclusion

Behavior representation language and behavior pattern detection and processing techniques are essential parts of user behavior analysis. User behavior is hidden within the transactional data and behavioral properties are separated from each other and separately recorded. Querying this kind of repositories relies on an appropriate language in order to represent the specification of the desirable behavior patterns. Moreover, a realistic dataset is required to evaluate the user behavior pattern analysis methods as well as behavior pattern matching approaches. However, the lack of real production datasets is the main motivation to develop an integrated framework to design a dataset with embedded user behavior patterns and a series of utilities to verify the generated dataset. In this case, the data analysts need a template to present the specification of the desirable behavior pattern.

In this paper, we proposed a generic Behavior Pattern Language (BPL) as a modeling language to represent attributes and properties of the behavior pattern as well as to represent relationships between the behavior entities. Basically, this language provides a generic template that can be utilized by analysts to represent the features of a user behavior pattern. Moreover, it can be used to represent the primitive elements and constraints of a specific pattern in order to construct behavioral data. The high level features and programming style of the BPL facilitate the representation of the behavior pattern in detail.

BPL is declarative enough to illustrate different user behavior patterns, while its programming-language syntax is human-understandable and easy to parse. BPL includes some built-in operators and statements for expressing particular operations based on an unambiguous grammar to define the structure and semantics of the behavior patterns. Based on the semantics of user behavior patterns, we categorized the behavior patterns into five groups such as: sequence, association, proximity, separation, and causality. The designed operators and features enable the BPL for representing such groups. We aim at extending the work by providing additional well-defined operators and features for language to represent more categories of user behavior patterns in different domains including user-system interactions. Meanwhile, making the proposed semantics executable helps us to evaluate the BPL expressions and it simplifies the evolution of the language. Moreover, an executable operational semantics is clearly reflected in the interpreter.

A The syntax of BPL

```
Syntax of BPL
<BPL> ::= BEGIN <pattern_specification> END
<pattern_specification>::=<Declaration><Operation>|
<Constraint>
<Declaration>::= <event declaration>|<pattern
declaration>|<set declaration>|<attVal declaration>
<event declaration>::= DEFINE Event    =
   <    {<atttName>}+    >
<pattern declaration>::=<leftside>    =    <event type>
<event type>::= NEW Event( <Nnumber> )
<set declaration>::= <leftside>    =       {    {<eventIdentifier
    >}+    }
<attVal declaration>::= <leftside>    =       {    {<valuelist>}+
       }
<valuelist>::= <attName> {<value>}+
<leftside>::= <identifier> | <identifier>[]
<Operation>::= <operation header> <operation body>
<operation header>::= DEF <identifier> {<parametrs>}
<operation body>::= <statements> END
<statements>::= <foreach statement>|<ifstatement>|
<yield statement>|<block statement>|<call statement>
<yield statement>::= yield { <parameters>}
<block statement> ::= DO { <parameters>} <statements> END
<call statement>::= CALL <method name> { <parameters>}
[<block statement>]
<foreach statement>::= FOREACH <identifier> IN
<identifier> DO
<ifstatement>::= IF < expression> THEN <statements>
<Constraint>::= <ConstraintExpr>|<call statement>
<ConstraintExpr>::= <expression>
< comparable operator><expression>
<comparable operator> ::=    <   |   >   |   =   |    !=   |   <=
     |   >=
<parametrs>::= set | event | <value>
< atttName>::= <identifier>
<identifier>::= a string of character
<eventIdentifier>::= event id
<Nnumber>::= a positive integer value
<value>::= integer | float | string
```

References

1. Alvarez M (2015) Battling security threats from within your organization. In: Research Report, IBM Security
2. Angeletou S, Rowe M, Alani H (2011) Modelling and analysis of user behaviour in online communities. Springer Verlag Berlin Heidelberg, pp. 35–50
3. Anicic D, Fodor P, Rudolph S, Stuhmer R, Stojanovic N, Studer R (2010) A rule-based language for complex event processing and reasoning. In: Web reasoning and rule systems: Fourth International Conference, Springer-Verlag Berlin Heidelberg, pp. 42–57
4. Arasu A, Babu S, Widom J (2006) The cql continuous query language: semantic foundations and query execution. The International Journal on Very Large Data Bases 15(2):121 – 142
5. Aztiria A, Augusto J C, Basagoiti R, Izaguirre A, Cook D J (2013) Learning frequent behaviors of the users in intelligent environments. IEEE Transactions on Systems, Man and Cybernetics, vol 43, no 6 pp. 1265–1278
6. Barga R S, Goldstein J, Ali M, Hong M (2007) Consistent streaming through time: A vision for event stream processing. In: 3rd Biennial Conference on Innovative Data Systems Research (CIDR), California
7. Brenna L, Demers A, Gehrke J, Hong M, Ossher J, Panda B, Riedewald M, Thatte M, White W (2007) Cayuga: a high-performance event processing engine. In: ACM SIGMOD international conference on Management of data, ACM, pp. 1100 – 1102
8. Bry F, Eckert M (2006) A high-level query language for events. In: IEEE Services Computing Workshops(SCW'06), IEEE
9. Bui H-L (2009) Survey and comparison of event query languages using practical examples. Ludwig Maximilian University of Munich
10. Cao L (2010) In-depth behavior understanding and use: the behavior informatics approach. Journal of Information Sciences, vol 180, no 17 pp. 3067- 3085
11. Cao L (2014) Behavior informatics: A new perspective. IEEE Intelligent Systems pp. 62–80
12. Cugola G, Margara A (2012) Processing flows of information: From data stream to complex event processing. ACM Computing Surveys (CSUR) 44(3):1 – 62
13. D Anicic S R P Fodor, Stojanovic N (2011) Ep-sparql: A unified language for event processing and stream reasoning. In: in Conference on World Wide Web , Hyderabad
14. Fernndez M (2014) Programming Languages and Operational Semantics, A Concise Overview. Springer-Verlag London
15. Fox A, Patterson D (2013) Engineering Software as a Service: An Agile Approach Using Cloud Computing. Strawberry Canyon, LLC
16. Grieskamp W, Kicillof N (2006) A schema language for coordinating construction and composition of partial behavior descriptions. In: SCESM06, Shanghai
17. Kirou A, Ruszczycki B, Walser M, Johnson N (2008) Computational modeling of collective human behavior: The example of financial markets. Springer Verlag Berlin Heidelberg, pp. 33–41
18. Li G, H, Jacobsen A (2005) Composite subscriptions incontent-based publish/subscribe systems. In: Middleware 2005: The series Lecture Notes in Computer Science, Springer-Verlag NewYork, pp. 249–269
19. Luckham D (2012) Event Processing for Business: Organizing the Real-Time Enterprise. New Jersey: Wiley
20. Pietzuch P R, Shand B, Bacon J (2004) Composite event detection as a generic middleware extension. IEEE Network Journal 18(1):44 – 55
21. Plotkin G (1981) A structural approach to operational semantics. Technical Report, Aarhus University, Denmark
22. Priya R V, Vadivel A (2012) User behaviour pattern mining from weblog. International Journal of Data Warehousing and Mining, vol 8, no 2 pp. 1–22
23. Python (2015) The python programming language website Https://www.python.org/
24. Rieke R, Zhdanova M, Repp J, Giot R, Gaber C (2013) Fraud detection in mobile payments utilizing process behavior analysis. In: IEEE International Conference on Availability, Reliability and Security, IEEE, pp. 946 – 953
25. Rozsnyai S, Schiefer J, Roth H (2009) Sari-sql: Event query language for event analysis. In: IEEE Conference on Commerce and Enterprise Computing, IEEE
26. Sandell N F, Savell R, Twardowski D, Cybenko G (2009) Hbml: A language for quantitative behavioral modeling in the human terrain. In: Social Computing and Behavioral Modeling,

Springer, pp. 180–189
27. Stolfo S, Bellovin S, Hershkop S, Keromytis A, Sinclair S, Smith S (2008) Insider attack and cybersecurity: Beyond the hacker. Springer, New York
28. Thomas D, Fowler C, Hunt A (2013) Programming Ruby 1.9 and 2.0, The Pragmatic Programmers. LLC
29. Wang C, Cao L (2012) Modeling and analysis of social activity process. Behavior computing: modeling, analysis, mining and decision, New York, Springer, pp. 21–35
30. Wooldridge M (2000) Reasoning about rational agents. Cambridge: MIT Press
31. Yarmand M, Sartipi K, Down D (2013) Behavior-based access control for distributed healthcare systems. Journal of Computer Security pp. 1–39
32. Zarri G P (2012) Behaviour representation and management making use of the narrative knowledge representation language. In: Behavior Computing, London, Springer-Verlag, pp. 37–56
33. Zerkouk M, Mhamed A, Messabih B (2013) User behavior and capability based access control model and architecture. In: Springer Science and Business Media, Springer Science and Business Media, pp. 291–299