Softw Syst Model (2003) 2: 134–149 / Digital Object Identifier (DOI) 10.1007/s10270-003-0024-z

EB^3: an entity-based black-box specification method for information systems

M. Frappier, R. St-Denis

Département de mathématiques et d'informatique, Université de Sherbrooke, Sherbrooke (Québec) Canada J1K 2R1;
E-mail: Marc.Frappier@dmi.usherb.ca

Received: 4 May 2002/Accepted: 29 March 2003
Published online: 3 June 2003 – © Springer-Verlag 2003

Abstract. This paper describes a formal method for specifying the observable (external) behavior of information systems using a process algebra and input-output traces. Its notation is mainly based on the entity concept, borrowed from the Jackson System Development method, and integrated with the requirements class diagram to represent data structures and associations. The specification process promotes modular and incremental description of the behavior of each entity through process abstraction, entity type patterns, and entity attribute function patterns. Valid system input traces result from the composition of entity traces by using parallel composition operations. The association between input traces and outputs through an input-output relation completes the specification process.

Keywords: Trace-based specifications – Black-box specifications – Process algebra – JSD – Cleanroom – Patterns

1 Introduction

An information system (IS) can be viewed as a set of related entities that produce information in response to user requests (operations) under the supervision of a decision-making procedure in order to manage the entities according to specific business rules. During its execution, the IS makes decisions based on its current internal state to ensure that the business rules are enforced. This paper is concerned with organizing unstructured streams of operation-response pairs with the sole aim of producing a complete, formal specification of the external system behavior, rather than a representation of design internals imposed by information system technologies. Instead of focusing on the decision-making procedure, it describes a formal method, called EB^3, for structuring and writing down a trace-based specification. Since its first proposal in 1998 [14], this method has been improved after conducting several case studies both in academia and industry (e.g., [15, 16]).

An EB^3 specification consists of a) a requirements class diagram (called a business model), which defines entity types with their external events, attributes, and associations; b) entity type specifications, which define the scenarios (valid sequences of input events) of entities using process expressions; c) entity attribute definitions, which are recursive functions on the system trace; and d) input-output rules, to specify outputs for input traces, or SQL expressions, which can be used to specify queries on the business model.

EB^3 is scenario-oriented and more abstract than state-transition specifications (STS), which is the most widely used paradigm for specifying information systems. An STS consists of a state space, defined by state variables, and operations describing transitions by modifications of the state variables. Upon reception of an external input event, an operation is called to compute the new state and to produce an output, if necessary. There exist several notations for describing STS, e.g., extended state machines, UML with OCL, model-based notations like B Z, VDM. One difficulty with STS is the validation and understanding of event ordering properties. Because ordering is expressed through conditions on state variables, it cannot be easily analyzed by human inspection. Because state variables constitute an encoding of the event history, the human inspector must have a very good understanding of the state transformations conducted in operations in order to validate these properties. In contrast, EB^3's entity process expressions provide an explicit representation of ordering constraints. Attribute definitions are *encapsulated* in a single expression, which facilitates their understanding and maintenance. We believe that this locality of information can simplify system understanding and streamline main-

Correspondence to: M. Frappier

tenance. Generally, adding an attribute to an entity does not induce any change to the rest of the EB^3 specification; its recursive function needs simply to be written on the system trace. In STS, each operation that affects the value of the attribute must be modified.

Another salient feature of EB^3 is its executability. We are currently working on tools to efficiently execute an EB^3 specification. The first tool is a process algebra interpreter that can execute a large class of process expressions in $O(s+\log(n))$ in time and $O(s+n)$ in space, where s is the size of the specification text and n is the number of entities [11, 12]. Typically, the complexity of a manual implementation of an IS specification is $O(\log(n))$ in time and $O(n)$ in space. Therefore, this interpreter has an overhead of $O(s)$ compared with a manual implementation of an IS. We are also working on algorithms to generate an SQL database schema and update operations in Java from EB^3's attribute definitions. The ultimate goal is to produce an efficient interpreter/code generator for EB^3 specifications that would be as good as conventional implementations. It therefore would relieve software engineers from implementing the specification. This goal seems quite feasible for a large subset of simple information systems [18].

The specification method advocated in this paper does not invent totally new concepts. Our main contribution is to adapt and uniformly integrate concepts from several methods that have already been successfully used in industrial applications: the black-box specification of Cleanroom [24, 27, 28], the entity structure diagram of the Jackson System Development (JSD) method [6, 21], the entity-relationship diagram [7], and theories that have sound mathematical foundations (e.g., process algebras [19], calculus of concurrency and synchronization [26], regular languages, first-order logic).

Writing complete, precise, and concise Cleanroom black-box specifications is not an easy task. Several practitioners use informal, or semi-formal, descriptions (e.g., [23]), but they tend to be quite long and their informality may lead to ambiguities. Some formal approaches have been proposed (e.g., [3, 13, 29]), but they may also lack conciseness because of the combinatorial explosion due to the inductive nature of their notation.

Several approaches studied the expression of ordering constraints using process algebras or regular expressions. Sridhar et al. [31] propose a pure CSP formalization of JSD entity structure diagrams. Both inputs and outputs are specified using CSP processes. No distinction is made between inputs and outputs, aside from CSP's decoration "?" and "!" on channels. Entity attributes are represented as process expressions. In contrast, we use a process algebra to specify input traces and input ordering constraints. Data structures and outputs are explicitly represented through a class diagram and recursive functions on traces, which makes it easier to represent the complex relationships found in data structures of information systems and to express queries using a standard notation like SQL. In [34], a formalization based on CSP and denotational semantics is proposed for *all* steps of the JSD method. Our approach is based only on the entity structure step of JSD.

The work in [25] focuses on unrelated operational functions seen as an ordered grouping of unit tasks without attributes, rather than related entities seen as an ordered grouping of operations with attributes. Ordering constraints are expressed by using regular expressions; operational functions are combined by using the interleave operation. Hsia et al. [20] proposed a scenario-based method in which scenarios constitute sentences of a regular language defined by a regular grammar.

To bring out the differences between these approaches and EB^3, a trace-based specification can be divided into several layers, from the least specific to the most specific descriptions, as summarized in Table 1. Each layer provides a different viewpoint and adds information to the previous layer. The table indicates what kind of business rules are scrutinized and if inputs or outputs are taken into account in each particular layer. The approaches in [31] and [34] cover the first four layers; [25] and [20] are less powerful than the one proposed herein in the sense that they only partially cover the first three layers. In [25], the method only deals with local ordering constraints since operational functions are executed independently. In [20], scenarios are considered as flat or unstructured sequences of events, since it deals only with global ordering constraints. Moreover, it is suitable for systems that have a single response to a stimulus, both being events without attributes.

Our specification method is related to recent work on scenarios since trace-based specifications provide, for instance, a good alternative to the precise description of

Table 1. Specification layers and their corresponding behavior

Layer	External behavior	Business rules	Input	Output
1	Event signatures	Description of input and output events	✓	✓
2	Entity behavior	Ordering constraints on each entity	✓	
3	System behavior	Ordering constraints on the system	✓	
4	Input-output behavior	Input-output rules	✓	✓
5	Robust behavior	Unspecific messages on wrong inputs	✓	✓
6	Final behavior	Specific messages on wrong inputs	✓	✓

use cases [5]. They satisfy the abstraction goal that use case proponents pursue, without requiring the definition of objects and classes as interaction diagrams do. Furthermore, this abstraction level does not hinder the definition of an object-oriented model, because entity types are naturally refined to become classes. Entities are also more expressive than interaction diagrams, because they provide complete description of system scenarios. The amount of recent work on scenarios in the software engineering community is overwhelming; see [1, 22, 33] for literature surveys on this subject.

The rest of the paper is organized as follows. Section 2 introduces the EB^3 specification method with an example to help the reader better understand the main concepts of its formal language as well as each phase of its specification process. Section 3 describes the syntax and semantics of a process algebra that is used in trace-based specifications. Section 4 summarizes the formal semantics of the EB^3 language. Section 5 introduces several patterns written in the EB^3 language that can be reused as building blocks in the specification of an IS. Finally, Sect. 6 concludes with a discussion of the strengths and limitations of EB^3, and prospects for future work.

2 The EB^3 specification method

The EB^3 specification method is specifically devised to derive the input-output behavior of an IS. Following Davis [8], one may distinguish between specification of static software systems and specification of dynamic software systems. Static systems (e.g., a compiler) mimic a simple input-output behavior, where the output is determined by the current input. Dynamic systems (e.g., an operating system) simulate an input-output behavior, where the current output is determined not only by the current input but also by the *history* of past inputs (input traces). An information system composed of distinct, but related, entities is considered as a dynamic system and is described by a new kind of black-box specification.

The core of EB^3 includes a specification process and a formal notation for progressing from use case diagrams to a complete and precise specification of input-output traces. Under EB^3, the specifier performs various tasks summarized as follows:

1. Define a business model.
2. Declare input-output signatures for entity types and associations.
3. Specify behavior of entity types and associations using process expressions.
4. Describe the set of valid system input traces.
5. Write recursive functions on traces that assign values to entity attributes.
6. Define an output for each input trace using input-output rules.

To illustrate the specification method, a subset of an IS that processes data about patients in an hospital is used. A patient is admitted to an hospital ward to receive treatment and medication. Medical acts are recorded in the patient's file, which is opened upon initial admission. A ward can only receive a limited number of patients.

2.1 The business model

The task of specifying system traces is made easier by decomposing a system into entity types. Based on UML [5], a business model is represented by a requirements class diagram that contains the entity types and their associations. In contrast to a UML design class diagram that includes attributes and methods for each class or relationship, a requirements class diagram identifies attributes and inputs of each entity type or association. An input results from an action that occurs in the environment (e.g., an action taken by a user or another system). It corresponds to an indivisible (or atomic) operation that an IS can perform. In EB^3, the terms *entity type* and *entity* are used instead of class and object. Furthermore, the term *process* will also be used in place of entity when an instance of an entity type is considered as a dynamic object. Entity types, associations, and their respective attributes are usually translated into database tables that the environment can query and update using an IS.

Figure 1 illustrates the requirements class diagram for the hospital admissions system. There are two entity

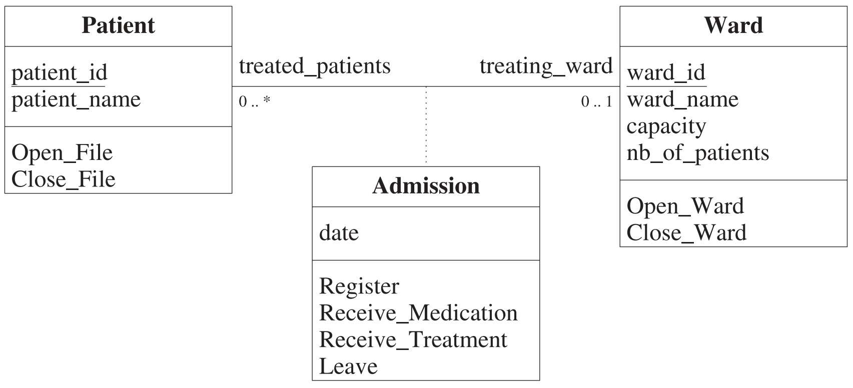

Fig. 1. The requirements class diagram of the hospital admissions system

types (Patient and Ward) and one association (Admission). They are identified from use cases (omitted in this paper). A set of attributes whose values uniquely identify each entity is called a key. These attributes are underlined in the diagram.

2.2 The input-output signatures

A black-box specification is derived from an input space and an output space. These spaces are defined by the assumption that an IS inexorably produces an output after accepting an input and before processing the next input. This assumption allows for the implementation of a black-box specification using parallel transactions processed by a database management system with proper concurrency control mechanisms (e.g., two-phase locking or transaction serialization). When the system does not produce a visible output for a given input, the special value void is used. Typically, update operations deliver an invisible output or a confirmation message, since their only effect is to modify the internal system state, which is not visible to the user. To display information about the system, the user typically invokes inquiry operations, which normally provide a response without modifying the system's internal state.

2.2.1 Input-output pairs

The signature of an input-output pair has the form *input* : *output* detailed as follows:

$$\mathsf{action}(in_1 : T_1, \cdots, in_m : T_m) : \\ (out_1 : T_{m+1}, \cdots, out_n : T_{m+n}) \,,$$

where action is a label associated with the input, in_i $(1 \leq i \leq m)$ denotes an input parameter name, out_i $(1 \leq i \leq n)$ denotes an output parameter name, and T_i $(1 \leq i \leq m+n)$ is a set or identifier that denotes a set, which provides a type for a parameter. Therefore, each input-output pair defines an input set X_{action} (the Cartesian product of $\{\mathsf{action}\}$ and T_i, $1 \leq i \leq m$) and an output set Y_{action} (the Cartesian product of T_i, $m+1 \leq i \leq m+n$). The special value void is a shorthand for the output set $\{\mathsf{void}\}$. The input sets (resp. output sets) are grouped together to constitute the *input space* X (resp. *output space* Y). The following declarations define the input and output spaces of the hospital admissions system. Each input introduced in the business model in Fig. 1 is thus formally defined.

Register($patient\_id$: **PATIENT**, $ward\_id$: **WARD**,
    $date$: **DATE**) : void
Receive_Medication($patient\_id$: **PATIENT**,
    $medication$: **MEDICATION**) : void
Receive_Treatment($patient\_id$: **PATIENT**,
    $treatment$: **TREATMENT**) : void
Leave($patient\_id$: **PATIENT**) : void
Open_File($patient\_id$: **PATIENT**,
    $patient\_name$: **STRING**) : void
Close_File($patient\_id$: **PATIENT**) : void
Open_Ward($ward\_id$: **WARD**,
    $ward\_name$: **STRING**,
    $capacity$: **NAT**) : void
Close_Ward($ward\_id$: **WARD**) : void
Patient_on_Ward($patient\_id$: **PATIENT**,
    $ward\_id$: **WARD**) :
    ($result$: {Yes, No})
List_Patients_on_Ward($ward\_id$: **WARD**) :
    ($report$: **ONE_LEVEL_REPORT**)
List_Patients( ) :
    ($report$: **TWO_LEVEL_REPORT**)

2.2.2 Structured Input and Output

Sets that provide types for parameters are defined from elementary sets (e.g., **BOOL**, **NAT**, and **STRING**) and set operations. This allows for the specification of a structured input or a structured output. Some useful set operations are the Kleene closure (A^*) and positive closure (A^+), which represent a list of elements of A and a nonempty list of elements of A, respectively. A set A may also be defined using the Cartesian product, that is expressed by a list $(c_1 : S_1, \cdots, c_n : S_n)$, where c_i is a coordinate name and S_i is a set expression. An elementary set that appears in a set expression is assumed to be defined. As an example, consider the following declarations that describe a two-level occupancy report by patient and ward:

TWO_LEVEL_REPORT $\triangleq$ ($title$: **STRING**,
    $wards$: **FIRST_LEVEL***,
    $grand\_total$: **NAT**)
FIRST_LEVEL $\triangleq$ ($ward\_name$: **STRING**,
    $patients$: **SECOND_LEVEL***,
    $total$: **NAT**)
SECOND_LEVEL $\triangleq$ ($patient\_id$: **PATIENT**,
    $patient\_name$: **STRING**)

Several functions are available to extract elements of an input or output. Parameter names and coordinate names are projection functions. They can be combined together with functions on lists for accessing an element included in a structured input or structured output. As an example, the term

$$patient\_name(first(patients \\ (first(wards(report(o))))))$$

refers to the name of the first patient on the first ward that appears in an output o of type **TWO_LEVEL_REPORT**. The subterm $wards(report(o))$ extracts the list of all wards, which constitutes the first level of the report. The subterm $first(wards(report(o)))$ gives the first ward of the previous list.

2.3 The behavior of entity types and associations

An entity in EB^3 is represented by a trace consisting of the inputs related to this entity. For instance, patient p_1 can be represented by the following entity trace:

$$\begin{aligned}\mathsf{Open\_File}(p_1, Paul)\,\textbf{.}\,&\mathsf{Register}(p_1, w_1, 26.4.02)\,\textbf{.}\\ &\mathsf{Receive\_Medication}(p_1,\\ &\qquad morphine)\,\textbf{.}\\ &\mathsf{Receive\_Treatment}(p_1,\\ &\qquad blood\,transfusion)\,\textbf{.}\\ &\mathsf{Leave}(p_1)\,\textbf{.}\\ &\mathsf{Register}(p_1, w_2, 27.4.02)\,\textbf{.}\\ &\mathsf{Receive\_Treatment}(p_1,\\ &\qquad major\ surgery)\end{aligned} \qquad (1)$$

This entity trace contains seven inputs related to patient p_1. Inputs are concatenated using operation "**.**" to form a trace. The aforementioned trace indicates that the hospital opened a file for patient Paul. Next, this patient was registered on ward w_1 (e.g., the trauma unit) and received morphine and a blood transfusion. Then, he was transferred from the trauma unit to ward w_2 (e.g., the surgical unit). He has undergone major surgery and has not left this unit yet.

2.3.1 Definition of entity types

The behavior of an IS is derived from the set of all permissible input traces for each entity. This is achieved by first imposing ordering constraints on the inputs specific to each entity type by using process expressions (PEs) whose syntax is very similar to regular expressions. PEs can also be seen as a scenario specification. Specifying ordering constraints on the inputs specific to each entity type fosters a step-wise construction of the requirements model. The following definition gives the ordering constraints for the entity type Patient.

$$\texttt{Entity Type}\ Patient(pId : \mathbf{PATIENT}) \triangleq$$
$$\mathsf{Open\_File}(pId, \_)\,\textbf{.}\,Admission(pId, \_)^*\,\textbf{.}\,\mathsf{Close\_File}(pId)$$

The body of this definition is a PE that specifies the set of all permissible complete traces of the patient with key pId. The first input of these traces is Open_File. The symbol "_" denotes an arbitrary choice of a value from the type of the second input parameter (i.e., *patient_name*). The first input must be followed by a sequence of inputs generated by the call to process *Admission*, which will be defined in the Sect. 2.3.2. This process defines the ordering constraints of association Admission in the business model. Operation "*" permits any number of admissions for a particular ward. Finally, the last input is Close_File.

Inquiry operations that cannot be associated with specific entities are grouped together into a single entity type, Query, which is not shown in the requirements class diagram in Fig. 1. Inputs that correspond to inquiry operations initiated by a user may be submitted at any time since they typically do not have strong ordering constraints. They are generally specified by a choice between inquiry inputs, surrounded with the operation "*", as:

$$\begin{aligned}&\texttt{Entity Type}\ Query() \triangleq\\ &\quad(\mathsf{Patient\_on\_Ward}(\_, \_) \mid \mathsf{List\_Patients\_on\_Ward}(\_) \mid\\ &\quad\ \mathsf{List\_Patients}())^*\end{aligned}$$

The set of traces generated by a PE is all sequences of inputs that the process may execute. A trace need not be complete (i.e., it does not need to include all the inputs up to the end of the PE). For instance, the entity trace (1) is a trace that belongs to the set of traces generated by the process $Patient(p_1)$.

2.3.2 Definition of associations

Each association is defined by a process, which is exclusively invoked in the definitions of the corresponding entity types. The process for association Admission is defined below.

$$\begin{aligned}&\texttt{Association}\ Admission(pId : \mathbf{PATIENT},\\ &\qquad wId : \mathbf{WARD}) \triangleq\\ &\quad(nb\_of\_patients(\mathsf{t}, wId) < capacity(\mathsf{t}, wId) \Longrightarrow\\ &\qquad \mathsf{Register}(pId, wId, \_))\,\textbf{.}\\ &\quad(\\ &\qquad \mathsf{Receive\_Medication}(pId, \_)\\ &\quad\mid\\ &\qquad \mathsf{Receive\_Treatment}(pId, \_)\\ &\quad)^*\,\textbf{.}\\ &\quad\mathsf{Leave}(pId)\end{aligned}$$

An admission is represented by a sequence of inputs starting with a Register, followed by a sequence of inputs generated from a PE, and terminating with a Leave. The Boolean expression $nb\_of\_patients(\mathsf{t}, wId) < capacity(\mathsf{t}, wId)$ that appears just before the elementary PE $\mathsf{Register}(pId, wId, \_)$ is a guard. It indicates that a patient can only be registered if the ward is not full. In this expression, the attributes *nb_of_patients* and *capacity* are recursive functions on the set $X^* \times \mathbf{WARD}$. The first argument t is a global variable that denotes the current valid system input trace. Finally, the innermost PE represents a choice (operation "$|$") between Receive_Medication and Receive_Treatment. It is surrounded with the operation "*" that permits any number of medications and treatments.

2.3.3 Relationships between entity types

The other entity type of the hospital admission system is a ward. Its definition in terms of ordering constraints on its inputs is provided below.

$$\begin{aligned}&\texttt{Entity Type}\ Ward(wId) \triangleq\\ &\quad\mathsf{Open\_Ward}(wId, \_, \_)\,\textbf{.}\\ &\quad(|||\ pId : \mathbf{PATIENT} : Admission(pId, wId)^*)\,\textbf{.}\\ &\quad\mathsf{Close\_Ward}(wId)\end{aligned}$$

The PE for the entity type Ward is the most complex. It shows how patients are treated on a ward. It refers to the process *Admission*, because a ward admits patients. It also contains a quantified operation of the form "$||| \; x : X : p(x)$", which represents the interleave of a number of processes described by PE $Admission(pId, wId)^*$ for every element pId of **PATIENT**. In concrete terms, it expresses the fact that a ward may treat several patients concurrently in a very compact form.

There are important differences in the structures of the entity type Patient and entity type Ward with respect to admission scenarios. First, a patient is, at most, in one ward at a time, but he can move from one ward to another ward several times, since the admission scenarios for a patient are represented by the PE $Admission(pId, \_)^*$ in the entity type Patient. Second, a ward may process many admissions concurrently and admit the same patient several times. The PE $Admission(\_, wId)^*$ alone is not appropriate for ward traces, because it implies that a ward would treat only one patient at a time. This is the reason why a quantified interleave is used. It should be noted that the interleave of all elements of set **PATIENT** does not imply that a ward must treat all the patients that have a file in the hospital; the PE permits it, but it also accepts that some patients are never registered on a ward, because of the "*" operation in the PE $Admission(pId, wId)^*$. The following trace is an example of a permissible entity trace of ward w_1.

$$\begin{array}{l} \mathsf{Open\_Ward}(w_1, \mathit{trauma\ unit}) \,. \\ \quad \mathsf{Register}(p_1, w_1, 26.4.02) \,. \\ \quad \mathsf{Register}(p_2, w_1, 26.4.02) \,. \\ \quad \mathsf{Receive\_Treatment}(p_2, \mathit{electrocardiogram}) \,. \\ \quad \mathsf{Receive\_Medication}(p_1, \mathit{morphine}) \,. \\ \quad \mathsf{Leave}(p_2) \,. \\ \quad \mathsf{Receive\_Treatment}(p_1, \mathit{blood\ transfusion}) \,. \\ \quad \mathsf{Leave}(p_1) \end{array} \tag{2}$$

Two patients were in ward w_1, but they eventually left after receiving treatments or medications.

2.3.4 Entity trace versus system state

During the implementation phase of the hospital admissions system, an appropriate representation of the system state must be chosen. For instance, if a relational database is used, a patient could be represented by a tuple in a patient table. If a patient's history of admissions must be saved, it could be represented by a set of tuples in an admission table. If only the last admission needs to be saved, it could be represented as attributes in the patient table. Several other representations are possible. An entity trace is a very abstract representation of the system state for a particular entity. For instance, a patient entity trace represents a tuple in the patient table and associated tuples in the admission table (assuming the history of admissions is required). Traces have several advantages over more concrete representations like tuples of a relational database or objects of an object-oriented database. First, they are very resilient to requirements changes. Deciding to keep the history of admissions or simply the last admission requires no change to the definition of the entity type Patient. An entity trace contains all the inputs of a patient; hence, both the last admission or history of admissions can be determined from it.

2.4 The set of valid system input traces

The system input trace records the inputs in their arrival order. It represents a global view of the system, whereas an entity trace represents a local view of a specific entity. By examining entity traces (1) and (2), one can deduce that there are at least two other entities in the system: ward w_2, in which patient p_1 was registered, and patient p_2, who was registered on ward w_1. For the sake of simplicity, we assume there is no other entity. The entity traces for p_2 and w_2 are provided below.

$$\begin{array}{l} \mathsf{Open\_File}(p_2, \mathit{Mary}) \,. \\ \quad \mathsf{Register}(p_2, w_1, 26.4.02) \,. \\ \quad \mathsf{Receive\_Treatment}(p_2, \mathit{electrocardiogram}) \,. \\ \quad \mathsf{Leave}(p_2) \end{array} \tag{3}$$

$$\begin{array}{l} \mathsf{Open\_Ward}(w_2, \mathit{surgery\ unit}) \,. \\ \quad \mathsf{Register}(p_1, w_2, 27.4.02) \,. \\ \quad \mathsf{Receive\_Treatment}(p_1, \mathit{major\ surgery}) \end{array} \tag{4}$$

Because the definitions of entity types Patient and Ward both refer to the process *Admission*, some inputs occur in two entity traces. For example, the input Register(p_2, w_1, 26.4.02) appears in entity traces (2) and (3).

2.4.1 Assumptions under the construction of valid system input traces

Two assumptions are made for deriving valid system input traces from entity traces. First, an input that appears in many entity traces must occur only once in the system trace. Second, the order of inputs prescribed by entity types should be respected. The following trace is a valid system input trace; there are several other possibilities.

$$\begin{array}{l} \mathsf{Open\_Ward}(w_1, \mathit{trauma\ unit}) \,. \\ \quad \mathsf{Open\_Ward}(w_2, \mathit{surgery\ unit}) \,. \\ \quad \mathsf{Open\_File}(p_1, \mathit{Paul}) \,. \\ \quad \mathsf{Register}(p_1, w_1, 26.4.02) \,. \\ \quad \mathsf{Open\_File}(p_2, \mathit{Mary}) \,. \\ \quad \mathsf{Register}(p_2, w_1, 26.4.02) \,. \\ \quad \mathsf{Receive\_Treatment}(p_2, \mathit{electrocardiogram}) \,. \\ \quad \mathsf{Receive\_Medication}(p_1, \mathit{morphine}) \,. \\ \quad \mathsf{Leave}(p_2) \,. \\ \quad \mathsf{Receive\_Treatment}(p_1, \mathit{blood\ transfusion}) \,. \\ \quad \mathsf{Leave}(p_1) \,. \\ \quad \mathsf{Register}(p_1, w_2, 27.4.02) \,. \\ \quad \mathsf{Receive\_Treatment}(p_1, \mathit{major\ surgery}) \end{array} \tag{5}$$

2.4.2 Invalid system input traces

An invalid input trace is one that contains some wrong inputs that violate the assumptions. The following trace is not a valid system input trace. It results from the concatenation of traces (1) to (4) with duplicate inputs removed.

$$\begin{array}{l} \mathsf{Open\_File}(p_1, Paul)\,. \\ \quad \mathsf{Register}(p_1, w_1, 26.4.02)\,. \\ \quad \mathsf{Receive\_Medication}(p_1, morphine)\,. \\ \quad \mathsf{Receive\_Treatment}(p_1, blood\ transfusion)\,. \\ \quad \mathsf{Leave}(p_1)\,. \\ \quad \mathsf{Register}(p_1, w_2, 27.4.02)\,. \\ \quad \mathsf{Receive\_Treatment}(p_1, major\ surgery)\,. \\ \quad \mathsf{Open\_Ward}(w_1, trauma\ unit)\,. \\ \quad \mathsf{Register}(p_2, w_1, 26.4.02)\,. \\ \quad \mathsf{Receive\_Treatment}(p_2, electrocardiogram)\,. \\ \quad \mathsf{Leave}(p_2)\,. \\ \quad \mathsf{Open\_File}(p_2, Mary)\,. \\ \quad \mathsf{Open\_Ward}(w_2, surgery\ unit) \end{array} \qquad (6)$$

There are several problems with this trace. First, input $\mathsf{Register}(p_1, w_1, 26.4.02)$ occurs before input $\mathsf{Open\_Ward}$ $(w_1, trauma\ unit)$, which means that a patient was registered on a ward that was not open. A similar problem arises with the inputs $\mathsf{Register}(p_1, w_2, 27.4.02)$ and $\mathsf{Open\_Ward}(w_2, surgery\ unit)$. Second, the patient p_2 received treatment before his file was opened.

Of course, it should be possible to submit any input at any point in time during the execution of the system even though a ward is not open yet or the patient's file has not been processed yet. In these cases, the system should respond by issuing an appropriate error message and without modifying the internal system state. Hence, the IS should accept an invalid system input trace. It is, however, much simpler to deal with error messages in a subsequent phase of the specification process. In EB^3, the specification of scenarios is restricted to valid system input traces.

2.4.3 Formal specification of valid system input traces

The key to solve the problem of valid system input trace generation is to use the parallel composition operation "$\|$" inspired from CSP. This operation acts like a conjunction operation for process expressions. An expression $E_1 \,\|\, E_2$ has the following meaning. If E_1 (resp. E_2) can execute an input $a(\dots)$ such that action a does not occur in E_2 (resp. E_1), then E_1 (resp. E_2) can execute it alone. If E_1 can execute an input $a(\dots)$ such that a occurs in E_2, and vice-versa, then E_1 and E_2 must execute it simultaneously, and the input occurs only once in the trace. In other words, the processes E_1 and E_2 synchronize on shared actions.

As an example, the valid system input traces obtained by the composition of entity traces (1), (2), (3), and (4) are defined by the following PE[1]:

$$(\,(1)\,|||\,(3)\,)\,\|\,(\,(2)\,|||\,(4)\,) \qquad (7)$$

Traces (1) and (3) are interleaved, because there is no ordering constraint between two patient entity traces; ward entity traces (2) and (4) present a similar case. These interleave expressions are combined with the parallel composition operation, hence they synchronize on *Admission* inputs, which satisfies the two assumptions. As an example, trace (5) belongs to the set of traces generated by the PE (7).

In general, entity traces must be combined in an appropriate manner to form the set of valid system input traces. This set is constructed from PEs of entity types under some assumptions that reflect the relationships between entities as those formulated in Sect. 2.4.1. Hence, a possibility is to generalize the PE (7) as follows. Let $\mathbb{E} \triangleq \{e_1, \cdots, e_n\}$ be the set of entity types of an IS. The set of system input traces is given by the following PE:

$$\|\ e : e \in \mathbb{E} : |||\ k : K_e : e(k)^{0..1} \qquad (8)$$

All entities of the same type are interleaved. It does not matter in which order entities of the same type evolve because they are independent. The PE expression $e(k)^{0..1}$ means zero or one $e(k)$, that is, entity of type e with key k does not necessarily exist. The process expressions for each entity type are then composed in parallel, but they are synchronized on common inputs. As advocated in the JSD method, an input trace must satisfy all the ordering constraints specified for each entity.

2.5 Specifying entity attributes using recursive functions

Valid input traces are finite, but of arbitrary length. They may be scanned recursively by using functions that examine the last input and an appropriate action decided on. In EB^3, recursive functions are used to determine the value of attributes with respect to the current valid system input trace t; the specification of attributes is the method's fifth step.

For each entity type, there is one recursive function per attribute and one recursive function per association. As an example, the requirements class diagram in Fig. 1 reveals that the entity type Ward has four attributes and one association with the entity type Patient. Let $\mathbb{F}(A)$ denote the set of finite subsets of A. Five recursive functions are then defined as:

$$\begin{array}{rl} ward\_id : & X^* \to \mathbb{F}(\mathbf{WARD}) \\ ward\_name : & X^* \times \mathbf{WARD} \to \mathbf{STRING} \\ capacity : & X^* \times \mathbf{WARD} \to \mathbf{NAT} \\ nb\_of\_patients : & X^* \times \mathbf{WARD} \to \mathbf{NAT} \\ treated\_patients : & X^* \times \mathbf{WARD} \to \mathbb{F}(\mathbf{PATIENT}) \end{array}$$

[1] For the sake of concision, the trace number is used rather than the actual trace.

The functions *ward_name*, *capacity*, and *nb_of_patients* have the same domain ($X^* \times \mathbf{WARD}$), because they are non-key attributes of the entity type Ward. The domain of a function for a non-key attribute is always $X^* \times T_k$, where T_k is the type of the key in the entity; the range is the type of the attribute. A key attribute is represented differently. The domain of its function is simply X^*; the codomain is the finite power set of its type. For instance, *ward_id* is the key of the entity type Ward. Its corresponding recursive function returns the *subset* of **WARD** that represents the open wards in the system. Let expression $s' \vdash x$ denote the *right append* of element x to sequence s'. Function *ward_id* is expressed using a syntax inspired from CAML as:

```
$ward\_id(s) \triangleq$
  match $s$ with
    $\varepsilon$                                   $\rightarrow \emptyset$
    $s' \vdash \text{Open\_Ward}(wId, \_, \_)$    $\rightarrow \{wId\} \cup ward\_id(s')$
    $s' \vdash \text{Close\_Ward}(wId)$           $\rightarrow ward\_id(s') - \{wId\}$
    $s' \vdash \_$                               $\rightarrow ward\_id(s')$
```

The function *treated_patients* represents the association between a ward and patients. Following the convention used in UML, its name is given by the role name at the patient association end of the class diagram in Fig. 1. Since the multiplicity of this association end is "0..*", the function must return a subset of **PATIENT**. A dual function can be introduced for the entity type Patient to navigate from a patient to its ward. It is defined as:

$$treating\_ward : X^* \times \mathbf{PATIENT} \rightarrow \mathbf{WARD}$$

The following definition of *nb_of_patients* illustrates the use of *treating_ward*.

```
$nb\_of\_patients(s, wId) \triangleq$
  match $s$ with
    $\varepsilon$                                   $\rightarrow \perp$
    $s' \vdash \text{Open\_Ward}(wId, \_, \_)$    $\rightarrow 0$
    $s' \vdash \text{Close\_Ward}(wId)$           $\rightarrow \perp$
    $s' \vdash \text{Register}(\_, wId, \_)$      $\rightarrow 1 + nb\_of\_patients$
                                          $(s', wId)$
    $s' \vdash \text{Leave}(pId)$                 $\rightarrow$
                        if $treating\_ward(s', pId) = wId$
                        then $nb\_of\_patients(s', wId) - 1$
                        else $nb\_of\_patients(s', wId)$
    $s' \vdash \_$                               $\rightarrow nb\_of\_patients(s', wId)$
```

The range of a recursive function is implicitly augmented with a distinguished element $\perp$, denoting undefinedness. It should be noted how encapsulated is the definition of *nb_of_patients* compared with a state-based implementation (or state-based specification). In a state-based implementation, attribute *nb_of_patients* would be modified in three places: the method that handles an Open_Ward transaction (to be initialized with the value zero) and methods that handle Register and Leave transactions (to be incremented or decremented by one).

2.6 The outputs

The previous tasks of the specification process focus on local and global ordering constraints that must be satisfied by input traces. The only restriction imposed on outputs is that an input from an input set must produce an output from the corresponding output set. The final task of the specification process concerns the definition of a relation R that imposes more restrictions on outputs with respect to valid input traces. In the sequel, $s \lhd R \rhd y$ denotes that the pair $\langle s, y \rangle$ is an element of relation R. For instance, consider the following one element system trace: $\text{Open\_Ward}(w_1, \textit{trauma unit})$. According to the list of input-output pairs (see Sect. 2.2.1), the output produced is void, which means that no visible output is sent to the environment. Hence, $\langle \text{Open\_Ward}(w_1, \textit{trauma unit}), \text{void} \rangle \in R$, or $\text{Open\_Ward}(w_1, \textit{trauma unit}) \lhd R \rhd \text{void}$.

Let $\mathfrak{t}$ denote the current valid system input trace (5), then:

$$\mathfrak{t} \vdash \text{Patient\_on\_Ward}(John, w_1) \lhd R \rhd \text{No}$$
$$\mathfrak{t} \vdash \text{List\_Patients\_on\_Ward}(w_2) \lhd R \rhd \langle\ p_1 \text{ Paul }\rangle$$

In the first case, the input trace yields the output No, because patient *John* is not on ward w_1. In the second case, the input trace produces a list of patients on ward w_2. Patient *Mary* does not appear on this list, because she was never registered on this ward, while patient *Paul* has not left the surgical unit yet.

Three techniques are used to precisely define what is the output associated with a valid system input trace, according to the output's characteristics: functions, *SQL* statements, and report generators. It would have been possible to use only a full process algebra like LOTOS, which has the same computational power as the Turing machine, or a predicate calculus to specify the relationship between inputs sequences and outputs, but the use of commercial tools close to those used by practitioners streamlines this task.

2.6.1 Specifying outputs using functions

For input Patient_on_Ward, the following function can be used to determine if a particular patient is on a given ward:

```
$on\_ward(s, pId, wId) \triangleq$
  if $wId = treating\_ward(s, pId)$
    then return Yes
  else
    return No
```

This function can be used in a rule that specifies the output for an input Patient_on_Ward.

```
Rule $R_1$
  var $pId : \mathbf{PATIENT}$
      $wId : \mathbf{WARD}$
  input Patient_on_Ward($pId, wId$)
  output $on\_ward(\mathfrak{t}, pId, wId)$
endRule
```

Such a rule has a formal semantics defined in first-order logic. For instance, the rule R_1 has the following meaning:

$$R_1(\mathsf{t}, o) \Leftrightarrow \exists pId, wId: \\ pId \in \mathbf{PATIENT} \wedge wId \in \mathbf{WARD} \wedge \\ last(\mathsf{t}) = \mathsf{Patient\_on\_Ward}(pId, wId) \wedge \\ o = on\_ward(\mathsf{t}, pId, wId) \tag{9}$$

The term $last(s)$ denotes the last element of sequence s, provided s is not empty.

Generally, rules expressed using functions and first-order logic are suitable when the output refers to attributes (represented by recursive functions) or results from intensive calculations, because of a lack of appropriate attributes in the business model.

2.6.2 Specifying outputs by using *SQL* statements

In many cases, outputs can be expressed more easily by using SQL statements. To do so, we assume that the requirements class diagram is converted into a relational model. There exist classical algorithms for that task; they cover all cases of multiplicity for associations (1:1, 1:N, M:N) [10]. Hence, we can write SQL statements based on the tables generated from the requirements class diagram. To illustrate this concept, consider the input List_Patients_on_Ward. It can be defined with an SQL statement as:

```
Rule R2
  var wId : WARD
  input  List_Patients_on_Ward(wId)
  output select patient_id, patient_name
         from Patient
         where treating_ward = wId
endRule
```

2.6.3 Specifying outputs by using report generators

When outputs are too complex to be defined by recursive functions or SQL statements in a simple manner, a report generator is used. In that specific case, it is more appropriate to describe an output in a language familiar to programmers, because a specification written in a mathematical form is generally much too long and therefore useless. The following rule, in which the output is described in Oracle's SQL*PLUS, specifies the output for the input List_Patients.

```
Rule R3
  input  List_Patients()
  output TTITLE CENTER 'LIST OF PATIENTS BY WARD'
         BREAK ON ward_name
         COMPUTE COUNT LABEL TOTAL OF patient_id
                          ON ward_name
         COMPUTE COUNT LABEL 'GRAND TOTAL'
                            OF patient_id ON REPORT
         SELECT ward_name, patient_id,
                      patient_name
         FROM Ward, Patient
         WHERE Ward.ward_id = Patient.
                      ward_id (+)
         ORDER BY ward_name, patient_id;
endRule
```

3 The process algebra

Process algebras are powerful tools to express event ordering constraints. Furthermore, they permit significant leverage in reasoning about a system's properties or in building case tools for scenario generation and scenario validation. A suitable subset of CSP [19] and LOTOS [4] has been selected to facilitate the IS specification process. It corresponds, roughly speaking, to regular expressions plus parallel composition with synchronization. The syntax and semantics of the chosen operations have been simplified, and concepts such as nondeterminacy and the silent action (denoted by τ in CCS [26]) have been deliberately omitted.

Concatenation is used in EB^3 rather than the sequential composition in LOTOS, CCS, or CSP. It makes process expressions easier to write and read. For instance, it means that the following LOTOS expression $(\sigma_1; \mathbf{exit} \,|\, \sigma_2; \mathbf{exit}) \gg \sigma_3; \mathbf{stop}$ is simply written as $(\sigma_1 \,|\, \sigma_2) \centerdot \sigma_3$ in EB^3. Moreover, quantification over choice, interleave, and parallel composition streamlines the specification of complex process expressions.

3.1 Syntax

A process may be declared using a name P, a vector of typed formal parameters $\vec{q}$, and a body, which is a PE E. Its syntactical form is $P(\vec{q}) \triangleq E$. Process expressions are defined over a set of symbols Σ, called the input set. The special symbol λ denotes an internal action that a process may execute without requiring input from the environment. The alphabet of a PE E over Σ denotes the set of *action labels* used in a process expression; it is denoted by $\alpha(E)$ and defined as follows. Let Φ_1 a binary PE operation and Φ_2 be a unary PE operation, then $\alpha(\lambda) = \emptyset$, $\alpha(\boxdot) = \emptyset$, $\alpha(\mathsf{action}(t_1, \cdots, t_n)) = \{\mathsf{action}\}$, $\alpha(E_1 \Phi_1 E_2) = \alpha(E_1) \cup \alpha(E_2)$, and $\alpha(\Phi_2 E) = \alpha(E)$. By extension, $\alpha(\Sigma)$ denotes the set of action labels of the input set Σ. This definition of α is different from the one used in CSP and closer to the one used in LOTOS.

The PEs are defined recursively as follows. i) Elements of $\Sigma \cup \{\lambda\}$ represent *elementary* PEs. ii) The symbol $\boxdot$ is an elementary PE that represents a process that has completed its execution. iii) Let E, E_1, and E_2 be PEs over Σ, $n \in \mathbb{N}$, p be a predicate, $\vec{t}$ be a vector of terms, P a process name, and $\Delta \subseteq \alpha\Sigma$. The expressions $E_1 \centerdot E_2$, $E_1 \,|\, E_2$, E^*, E^+, $p \Longrightarrow E$, $P(\vec{t})$, $E_1 \,|[\Delta]|\, E_2$, $E_1 \,|||\, E_2$, and $E_1 \parallel E_2$ are PEs over Σ. Operations $\centerdot$, $|$, *, and $^+$ are the usual regular expression operations: concatenation, choice, Kleene closure, and positive closure. PE $p \Longrightarrow E$ is a guard; it states that E can execute an input only when p holds. PE $E_1 \,|[\Delta]|\, E_2$ is the parameterized parallel composition of E_1 and E_2 with synchronization on actions that belong to set Δ. Drawn from LOTOS, it has the following meaning: if E_1 (or, dually, E_2) can execute σ, and $\alpha(\sigma) \not\subseteq \Delta$, then the composition can execute σ. When $\alpha(\sigma) \subseteq \Delta$, then E_1 and E_2 must synchronize, that is, they must both execute σ. Operations $|||$ and $\|$ are the interleave and parallel composition of CSP, respectively; they are special variants of $|[\,]|$. The following precedence is imposed on operations, from highest to lowest: $\{^*, ^+\}$, $\boldsymbol{.}$, $\Longrightarrow$, $|$, $\{\,|[\,]|, |||, \|\}$, $\{\,|\,x,\ |||\ x,\ |[\,]|\,x\ \}$. It may be modified by means of parentheses.

3.2 Semantics of operations

In EB^3, a PE describes sequences of inputs that may be executed by a system. The meaning of PEs is described by the following operational semantics in the CCS style. Inference rules T-1 to T-16 define a transition relation $\rightarrow \subseteq \mathcal{PE} \times (\Sigma \cup \{\lambda\}) \times \mathcal{PE}$, where $\mathcal{PE}$ denotes the set of process expressions over Σ. Let ρ denote an element of $\Sigma \cup \{\lambda\}$.

(T-1): $\dfrac{\rho \in \Sigma \cup \{\lambda\}}{\rho \xrightarrow{\rho} \boxdot}$ (T-2): $\dfrac{p \wedge E \xrightarrow{\rho} E'}{p \Longrightarrow E \xrightarrow{\rho} E'}$

(T-3): $\dfrac{E_1 \xrightarrow{\rho} E_1'}{E_1 \boldsymbol{.} E_2 \xrightarrow{\rho} E_1' \boldsymbol{.} E_2}$ (T-4): $\dfrac{E \xrightarrow{\rho} E'}{\boxdot \boldsymbol{.} E \xrightarrow{\rho} E'}$

(T-5): $\dfrac{E_1 \xrightarrow{\rho} E_1'}{E_1 \,|\, E_2 \xrightarrow{\rho} E_1'}$ (T-6): $\dfrac{E_2 \xrightarrow{\rho} E_2'}{E_1 \,|\, E_2 \xrightarrow{\rho} E_2'}$

(T-7): $\dfrac{}{E^* \xrightarrow{\lambda} \boxdot}$ (T-8): $\dfrac{E \xrightarrow{\rho} E'}{E^* \xrightarrow{\rho} E' \boldsymbol{.} E^*}$

(T-9): $\dfrac{}{\boxdot \,|[\Delta]|\, \boxdot \xrightarrow{\lambda} \boxdot}$

(T-10): $\dfrac{E_1 \xrightarrow{\rho} E_1' \ \wedge\ E_2 \xrightarrow{\rho} E_2' \ \wedge\ \alpha(\rho) \subseteq \Delta}{E_1 \,|[\Delta]|\, E_2 \xrightarrow{\rho} E_1' \,|[\Delta]|\, E_2'}$

(T-11): $\dfrac{E_1 \xrightarrow{\rho} E_1' \ \wedge\ \alpha(\rho) \not\subseteq \Delta}{E_1 \,|[\Delta]|\, E_2 \xrightarrow{\rho} E_1' \,|[\Delta]|\, E_2}$

(T-12): $\dfrac{E_2 \xrightarrow{\rho} E_2' \ \wedge\ \alpha(\rho) \not\subseteq \Delta}{E_1 \,|[\Delta]|\, E_2 \xrightarrow{\rho} E_1 \,|[\Delta]|\, E_2'}$

(T-13): $\dfrac{p[x := t] \wedge E[x := t] \xrightarrow{\rho} E'}{|\, x : p : E \xrightarrow{\rho} E'}$

(T-14): $\dfrac{E[\vec{x} := \vec{t}\,] \xrightarrow{\rho} E'}{P(\vec{t}\,) \xrightarrow{\rho} E'}\ (P(\vec{x}) \triangleq E)$

(T-15): $\dfrac{\begin{array}{c} p[x := t] \wedge (\exists x : x \neq t \wedge p) \wedge \\ E[x := t] \,|[\Delta]|\, (\ |[\Delta]|\ x : p \wedge x \neq t : E) \xrightarrow{\rho} E' \end{array}}{|[\Delta]|\ x : p : E \xrightarrow{\rho} E'}$

(T-16): $\dfrac{\begin{array}{c} p[x := t] \wedge \neg(\exists x : x \neq t \wedge p) \wedge \\ E[x := t] \xrightarrow{\rho} E' \end{array}}{|[\Delta]|\ x : p : E \xrightarrow{\rho} E'}$

The following operations are defined as abbreviations: $E^+ \triangleq E \boldsymbol{.} E^*$, $E^{0..1} \triangleq E \,|\, \lambda$, $E_1 \,|||\, E_2 \triangleq E_1 \,|[\emptyset]|\, E_2$, $E_1 \,\|\, E_2 \triangleq E_1 \,|[\alpha(E_1) \cap \alpha(E_2)]|\, E_2$, $\mathsf{action}(\_) \triangleq |\, x : T : \mathsf{action}(x)$, where T denotes the type of parameter _ of action.

3.3 Traces

In order to define the set of traces associated with a PE, it is convenient to introduce the trace transition relation $\rightsquigarrow \subseteq \mathcal{PE} \times \Sigma^* \times \mathcal{PE}$. Let expression $\sigma \dashv s$ denote the *left append* of symbol $\sigma \in \Sigma$ to sequence $s \in \Sigma^*$. Inference rules TT-1 to TT-2 extend relation $\rightarrow$ by considering finite sequences s of elements of Σ rather than symbols. Note that λ is not included in traces.

(TT-1): $\dfrac{E \xrightarrow{\sigma} E'}{E \overset{\sigma}{\rightsquigarrow} E'}$ (TT-2): $\dfrac{E \xrightarrow{\sigma} E' \ \wedge\ E' \overset{s}{\rightsquigarrow} E''}{E \overset{\sigma \dashv s}{\rightsquigarrow} E''}$

(TT-3): $\dfrac{E \xrightarrow{\lambda} E' \ \wedge\ E' \overset{s}{\rightsquigarrow} E''}{E \overset{s}{\rightsquigarrow} E''}$

(TT-4): $\dfrac{E \overset{s}{\rightsquigarrow} E' \ \wedge\ E' \xrightarrow{\lambda} E''}{E \overset{s}{\rightsquigarrow} E''}$

A *trace* generated by a process described by $E \in \mathcal{PE}$ is a nonempty sequence $s \in \Sigma^+$ recording its evolution up to some moment in time. As usual, let ε denote the empty sequence. The set of traces associated with E, denoted $\mathcal{T}(E)$, is the set $\{s \mid \exists E' \in \mathcal{PE} : E \overset{s}{\rightsquigarrow} E' \wedge s \neq \varepsilon\}$.

Let $E_1, E_2 \in \mathcal{PE}$. PE E_1 is said to be *equivalent* to E_2, denoted $E_1 =_t E_2$, iff $\mathcal{T}(E_1) = \mathcal{T}(E_2)$. By using this definition, it can be shown that operations $|$, $|[\Delta]|$, $|||$, and $\|$ are commutative and associative under trace equivalence. Moreover, $\boxdot$ is the unit of $|$ and λ is the unit of $|||$ and $\|$. If $\alpha(E) \cap \Delta = \emptyset$, then $\lambda \,|[\Delta]|\, E =_t E$. Concatenation is associative and PE λ is the unit of this operation. Finally, all binary operations distribute through the choice operation.

4 The semantics of the EB^3 language

Recall that an EB^3 specification is composed of the following elements: a business model, input-output signatures, process expressions, recursive functions on traces, and input-output rules. To define the semantics of the EB^3 language, these five elements are mapped to an input-output relation R.

4.1 The Input-Output Relation

The input-output signature defines an input set and an output set that are denoted by X and Y, respectively. Relation R is a subset of $X^+ \times Y$. It is defined as follows. Let *main* denote the main process expression (e.g., (8)) and let R_i denote the predicate associated with an input-output rule (e.g., (9)). Relation R_{visible} defines the outputs of input traces using input-output rules.

$$R_{\text{visible}} \triangleq \{(\mathsf{t}, y) \mid \mathsf{t} \in \mathcal{T}(main) \\ \wedge (R_1(\mathsf{t}, y) \vee \cdots \vee R_n(\mathsf{t}, y))\}$$

Relation R_{void} defines the input-output pairs for input traces ending with an action whose output type is void (e.g., action $\mathsf{Open\_File}$). Let $Void$ denote the labels of such actions.

$$R_{\mathsf{void}} \triangleq \{(\mathsf{t}, \mathsf{void}) \mid \mathsf{t} \in \mathcal{T}(main) \wedge \alpha(last(\mathsf{t})) \in Void\}$$

The input-output relation defining the semantics of an EB^3 specification is:

$$R \triangleq R_{\text{visible}} \cup R_{\mathsf{void}}$$

4.2 Correctness

Defining how a specification relates to an implementation completes the semantics of the EB^3 language. A program is said to be a correct implementation of a specification R if, for any input trace $s = \sigma_1 \centerdot \cdots \centerdot \sigma_n \in dom(R)$, the program satisfies the following two properties: i) starting in its initial state, the program successively accepts each element of the input trace, from σ_1 to σ_n, delivering an output for each input; and ii) the output y produced for input σ_n satisfies $s \lhd R \rhd y$.

4.3 Robustness

An invalid input trace is one that contains a wrong input that requires special processing. A typical IS should display an appropriate message and process subsequent inputs as if this wrong input never happened. The input-output relation R associated with an EB^3 specification is not defined for such invalid input traces. The system behavior for invalid traces can be specified in a generic manner by defining a robust specification. Function *robust* takes a relation R between X^+ and Y and extends it to a *total* relation between X^+ and $(Y \cup$ **Messages**$)$, hence, a relation that is also defined for invalid traces.

$$\begin{aligned} robust(R) \triangleq \{(s' \vdash x, y) \in X^+ \times (Y \cup \mathbf{Messages}) \mid \\ (red(s', R) \vdash x \lhd R \rhd y \\ \vee \\ red(s', R) \vdash x \notin dom(R) \wedge y \in \mathbf{Messages})\} \end{aligned}$$

where function $red(s, R)$ is defined as follows:

$$\begin{array}{ll} \texttt{case} & \\ s = \varepsilon & \rightarrow \varepsilon \\ s \in dom(R) & \rightarrow s \\ s = s' \vdash x \ \wedge \ red(s', R) \vdash x \in dom(R) & \rightarrow red(s', R) \vdash x \\ s = s' \vdash \_ & \rightarrow red(s', R) \end{array}$$

Relation *robust*(R) ensures that at least one information message is associated with each invalid input trace. It does not, however, specify which one. Nevertheless, relation *robust*(R) can be refined by a relation, denoted R^f, that assigns specific information messages to invalid input traces. This can be done by using a technique similar to that proposed in Sect. 2.6. An input-output rule, where the output is an information message (msg) that belongs to **Messages**, is defined as:

$$s \vdash x \in X^+ \wedge s' = red(s, R) \wedge conjunction \\ \Rightarrow \ s \vdash x \lhd R^f \rhd msg.$$

Input trace s' is valid, but not $s' \vdash x$, because x represents a wrong input. This fact is reflected in conjuncts of *conjunction* in a form equivalent to the expression $s' \vdash x \notin dom(R)$. Relation R^f should be a total relation. It gives the *final* behavior of an IS.

5 Specification patterns

After conducting several case studies using the EB^3 method, both in a research environment and an industrial environment, we have noted that several patterns regularly occur in various specifications. They occur mainly in two places: i) in the process specification of the entity types and ii) in the definition of recursive functions.

5.1 Entity type patterns

Given a business model, one can easily select from a handful of patterns to generate the entity type process expression.

The producer-modifier-consumer pattern. Actions of an entity type can usually be classified as either producers, modifiers, or consumers (as in the SADT method). A producer creates an entity, a modifier changes the values of its attributes (as defined by recursive functions), and a consumer deactivates an entity. Association actions can also be classified in the same manner. The following pattern describes this producer-modifier-consumer structure in terms of a PE. Let e be an entity type of key set K_e and let $P^e_{ip}, M^e_{im}, C^e_{ic}$ denote a producer, a modifier, and a consumer, respectively, of entity type e. A first pattern for an entity is:

$$\begin{aligned} e(k : K_e) &\triangleq \mathcal{P}^e(k) \centerdot \mathcal{M}^e(k)^* \centerdot \mathcal{C}^e(k) \\ \mathcal{P}^e(k : K_e) &\triangleq P^e_1(k, \_) \mid \cdots \mid P^e_{np}(k, \_) \\ \mathcal{M}^e(k : K_e) &\triangleq M^e_1(k, \_) \mid \cdots \mid M^e_{nm}(k, \_) \\ \mathcal{C}^e(k : K_e) &\triangleq C^e_1(k, \_) \mid \cdots \mid C^e_{nc}(k, \_) \end{aligned}$$

The one-to-many association pattern. When entity type e_1 is related to another entity type e_2 by a one-to-many (1:N) association a, a typical pattern of their process expressions is:

$$
\begin{array}{ll}
e_1(k_{e_1} : K_{e_1}) \triangleq & e_2(k_{e_2} : K_{e_2}) \triangleq \\
\quad \mathcal{P}^{e_1}(k_{e_1}) \,. & \quad \mathcal{P}^{e_2}(k_{e_2}) \,. \\
\quad (& \quad (\\
\qquad \mathcal{M}^{e_1}(k_{e_1})^* & \qquad \mathcal{M}^{e_2}(k_{e_2})^* \\
\quad ||| & \quad ||| \\
\qquad ||| \; k_{e_2} : K_{e_2} : a(k_{e_1}, k_{e_2})^* & \qquad a(\_, k_{e_2})^* \\
\quad) \,. & \quad) \,. \\
\quad \mathcal{C}^{e_1}(k_{e_1}) & \quad \mathcal{C}^{e_2}(k_{e_2})
\end{array}
$$

On one hand, an entity of e_1 is related to many entities of e_2; hence, a quantified interleave on the association process a is used, denoting that associations are independent from each other. On the other hand, since an entity of e_2 is related to only one entity of e_1, a quantified choice is used, which is concisely expressed by using "_" in the process call $a(\_, k_{e_2})$. Note that a one-to-one association (1:1) can be represented by using two entity types of the e_2 style, and that a many-to-many association (M:N) can be represented by using two entity types of the e_1 style.

The multiple associations pattern. To generalize the previous pattern further, we may consider that an entity type may have several associations a_i with several entities. Let $\Theta(e_i, e_j, a, c)$ denote the pattern of associating entity type e_i with entity type e_j through association a with multiplicity c. Multiplicity c can either be 1 or N to denote that an entity of e_i is related to either one or many entities of e_j, respectively. Pattern Θ is defined as:

$$\Theta(e_i, e_j, a, 1) \equiv a(k_{e_i}, \_)^*$$

$$\Theta(e_i, e_j, a, N) \equiv ||| \; k_{e_j} : K_{e_j} : a(k_{e_i}, k_{e_j})^*$$

Pattern $\Theta(e_i, e_j, a, 1)$ states that an entity of e_i is related to at most one entity of e_j at any point in time. Furthermore, the entity to which it is related may change over time. For example, a patient may move from ward to ward during his stay, but he can be in only one ward at a time. Pattern $\Theta(e_i, e_j, a, N)$ provides that an entity of e_i may be related to several entities of e_j at any point in time. For instance, a ward can contain several patients; patients may leave and come back as often as needed. The following pattern describes how e_1 is related to other entity types through associations $a_1, \cdots, a_n$.

$$
\begin{array}{ll}
e_1(k_{e_1} : K_{e_1}) \triangleq & \mathcal{A}(k_{e_1} : K_{e_1}) \triangleq \\
\quad \mathcal{P}^{e_1}(k_{e_1}) \,. & \qquad \Theta(e_1, e_2, a_1, c_1^{e_1}) \\
\quad (& \qquad || \\
\qquad \mathcal{M}^{e_1}(k_{e_1})^* & \qquad \cdots \\
\quad ||| & \qquad || \\
\qquad \boxed{\mathcal{A}(k_{e_1})} & \qquad \Theta(e_1, e_{n+1}, a_n, c_n^{e_1}) \\
\quad) \,. & \\
\quad \mathcal{C}^{e_1}(k_{e_1}) &
\end{array}
$$

Note that the associations in process $\mathcal{A}$ are composed in parallel, because some associations may have actions in common. For instance, in a library system, there are two associations between entity type book and entity type member: loan and reservation. Loan and reservation have action BorrowWithReservation in common.

The n-ary association pattern. An association may involve more than two entity types. For instance, assume that e_1, e_2, e_3 are related through association a. The multiplicities that relate e_1 to e_2 and e_3 are mutually dependent; they cannot be handled two by two. As an example, consider the following definition of e_1:

$$
\begin{array}{l}
e_1(k_{e_1} : K_{e_1}) \triangleq \\
\quad \mathcal{P}^{e_1}(k_{e_1}) \,. \\
\quad (\mid k_{e_2} : K_{e_2} : ||| \; k_{e_3} : K_{e_3} : a(k_{e_1}, k_{e_2}, k_{e_3})^*)^* \,. \\
\quad \mathcal{C}^{e_1}(k_{e_1})
\end{array} \tag{10}
$$

$$
\begin{array}{l}
a(k_{e_1} : K_{e_1}, k_{e_2} : K_{e_2}, k_{e_3} : K_{e_3}) \triangleq \\
\quad P^a(k_{e_1}, k_{e_2}, k_{e_3}) \,. \\
\quad C^a(k_{e_1}, k_{e_2}, k_{e_3})
\end{array}
$$

It states that an entity of e_1 is related to at most one entity of e_2 and several entities of e_3. The inner closure allows for a pair of entities from e_1 and e_2 to re-establish their association with an entity of e_3. The outer closure allows for an entity of e_1 to replace its association with an entity of e_2. Hence, definition (10) allows for the following sequence, where we show only the association producer action P^a, which creates an association between e_1, e_2, e_3.

$$P^a(p_{1_1}, p_{2_1}, p_{3_1}) \,. P^a(p_{1_1}, p_{2_1}, p_{3_2})$$

In contrast, it does not allow for the following sequence:

$$P^a(p_{1_1}, p_{2_1}, p_{3_1}) \,. P^a(p_{1_1}, p_{2_2}, p_{3_2})$$

because it associates p_{1_1} with two entities of e_2, i.e., p_{2_1} and p_{2_2}. If the quantifiers in the definition of e_1 are interchanged, that is, if the following call to a is used:

$$||| \; k_{e_3} : K_{e_3} : \mid k_{e_2} : K_{e_2} : a(k_{e_1}, k_{e_2}, k_{e_3})^* \tag{11}$$

then the sequence above is accepted. The process expression of (11) states that an entity of e_1 can be related to several entities of e_3 and, for each entity of e_3, to exactly one entity of e_2, which is not the same behavior as expressed in (10). Hence, quantified process expressions are similar to quantified first-order formulas, where existential and universal quantifiers cannot be commuted without changing the meaning of a formula. Interleave quantifications can be commuted, however, like universal quantifiers in first-order logic.

The pattern for an n-ary association is not as easy to represent abstractly as the binary association pattern. Aside from the fact that the ordering of the quantifiers is significant, the specification of the Kleene closure (*) is also important. It allows for very fine distinctions between association behaviors. For instance, note that two

Kleene closures occur in (10) in order to freely choose and change the related entities while respecting the multiplicities, whereas only one closure is sufficient in (11). The most general and less restrictive pattern is:

$$(\Phi_2\, k_{e_2} : K_{e_2} : \;\cdots\; \Phi_n\, k_{e_n} : K_{e_n} : \\ a(k_{e_1}, k_{e_2}, \cdots, k_{e_n})^* \cdots)^* \,,$$

where Φ_i is either a choice (for a multiplicity of 1) or an interleave (for a multiplicity of N).

The weak entity type pattern. The notion of *weak entity type* is well known in database modeling. A weak entity type is dependent on another entity type, in the sense that it needs an entity of a parent entity type to exist. An item of a customer order is a typical example of a weak entity type. The order header must exist in order to add items. The parent entity is usually related to many weak entities. This pattern is represented as follows. Let we be a weak entity type (e.g., item) and let e be its parent entity type (e.g., order header). Usually, a weak entity type has producers, modifiers, and consumers, like a normal entity type.

```
$e(k_e : K_e) \triangleq$
  $\mathcal{P}^e(k_e)\,.$
  (
    $\mathcal{M}^e(k_e)^*$
  $|||$
    (
      $|||\; k_{we} : K_{we} : we(k_e, k_{we})^*$
    $||$
      $\mathcal{A}^e(k_e)$
    )
  ) .
  $\mathcal{C}^e(k_e)$
```

```
$we(k_e : K_e, k_{we} : K_{we}) \triangleq$
  $\mathcal{P}^{we}(k_e, k_{we})\,.$
  (
  $\mathcal{M}^{we}(k_e, k_{we})^*$
  $|||$
  $\mathcal{A}^{we}(k_e, k_{we})$
  ) .
  $\mathcal{C}^{we}(k_e, k_{we})$
```

The recursive association pattern. An association that relates an entity type with itself is usually called *recursive*. For instance, consider the manage relationship between employees. To state employee p_1 manages employee p_2, both p_1 and p_2 must have been created. This cannot be expressed by using a single entity type for employee; two are needed to represent both ends of the recursive association. Indeed, a recursive association is no different than a normal association: it needs two entity types to interact. Let e denote the process representing the entity type and e' denote this second process needed to represent the second end of the recursive association.

```
$e(k_e : K_e) \triangleq$
    $\mathcal{P}^e(k_e)\,.$
    (
        $\mathcal{M}^e(k_e)^*$
    $|||$
        $\Theta(e, e', a, c_e)$
    ) .
    $\mathcal{C}^e(k_e)$
```

```
$e'(k_e : K_e) \triangleq$
    $\mathcal{P}^e(k_e)\,.$
    $\Theta(e', e, a, c_{e'})\,.$
    $\mathcal{C}^e(k_e)$
```

Note that $\alpha(e') \subseteq \alpha(e)$. Hence, these two processes synchronize on producers, association actions and consumers in the *main* process (i.e., see (8)). Note there is no need to mention the modifiers in e', because their ordering constraints are already expressed in e. When an entity is created (e.g., an employee), both processes execute the same producer action. When an association producer action is executed (e.g., $\mathsf{Manage}(p_1, p_2)$), one entity in each entity type executes it. The example below of computing a system trace illustrates this. An action $P^e(p_1)$ denotes a producer of e, while the action $P^a(p_1, p_2)$ denotes an association producer action like $\mathsf{Manage}(p_1, p_2)$.

$$P^e(p_1)\,.\,P^a(p_1, p_2) \qquad (12)$$
$$||| \qquad (13)$$
$$P^e(p_2) \qquad (14)$$
$$|| \qquad (15)$$
$$P^e(p_1) \qquad (16)$$
$$||| \qquad (17)$$
$$P^e(p_2)\,.\,P^a(p_1, p_2) \qquad (18)$$
$$\rightsquigarrow$$
$$P^e(p_1)\,.\,P^e(p_2)\,.\,P^a(p_1, p_2) \qquad (19)$$

Line (19) denotes the system trace. Lines (12) and (14) denote the entity traces in entity type e; lines (16) and (18) denote the entity traces in process expression e'. Lines (13) and (17) denote the interleaving of entity traces of an entity type, while line (15) denotes the composition between entity types.

Furthermore, cycles are usually not allowed in recursive associations (e.g., p_1 manages $\cdots$ manages p_1). This constraint is better expressed by a guard:

$$k_{e'} \neq k_e \wedge k_{e'} \notin successors(\mathsf{t}, k_e) \implies P^a(k_e, k_{e'})$$

in PE a, where *successors* is an attribute of e defined as:

```
$successors(s, k_e) \triangleq$
  match $s$ with
    $\varepsilon$                      $\rightarrow \perp$
    $s' \vdash P^e(k_e, \_)$      $\rightarrow \emptyset$
    $s' \vdash P^a(k_e, k_1)$     $\rightarrow k_1 \cup successors(s', k_e)$
    $s' \vdash P^a(k_1, k_2)$     $\rightarrow$ if $k_1 \in successors(s', k_e)$
                             then $\{k_2\} \cup successors(s', k_e)$
                             else $successors(s', k_e)$
    $s' \vdash C^e(k_e)$          $\rightarrow \perp$
    $s' \vdash \_$                $\rightarrow successors(s', k_e)$
```

The inheritance association pattern. The inheritance association allows one entity type to inherit attributes and actions from another. A main pattern is presented and illustrated by the requirements class diagram in Fig. 2. It is founded on the following assumptions: a) single inheritance and no overloading, i.e., an entity type can only be a specialization of one entity type and an action occurs only once in the requirements class diagram; b) producers occur only on leafs of the inheritance

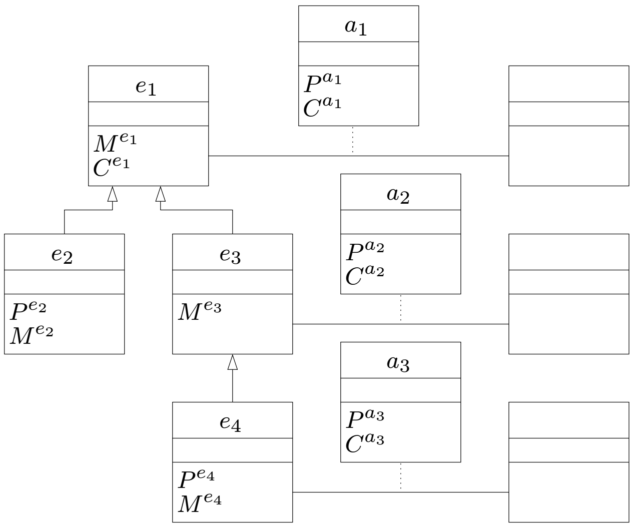

Fig. 2. An inheritance hierarchy in a requirements class diagram

hierarchy; c) any entity type can have modifiers and associations; d) only the top entity type has consumers; and e) the key is provided by the top entity type; hence, a key value must be unique among all entities of the inheritance hierarchy. Variations on these assumptions generate variations on the main pattern. The asymmetry between producers and consumers can be justified as follows: a producer from a generalization is usually not very useful for its specializations, because it does not provide values for their specific attributes. Dually, there is no need to have a consumer for each entity type in the hierarchy, because it usually has only one parameter: the entity type key, which is inherited, and whose value is unique among all entities of the hierarchy. Entity types that are neither a top or a leaf in the hierarchy only have modifiers and associations.

The main pattern consists of only one entity type process expression, e_g, which describes the ordering constraints for the entire inheritance hierarchy. Process e_g offers a choice between the leafs of the hierarchy, denoted by e_{s_i}, followed by its consumers. Each leaf calls its specific modifiers and associations as well as the modifiers and associations of its parent entity type. The following notation is used to handle inheritance: $\mathcal{M}^e_\star(k_e)$ denotes all modifiers accessible to entity type e; it includes the modifiers specific to e, denoted by $\mathcal{M}^e(k_e)$, and the modifiers inherited from its parent entity type, denoted by $\mathcal{M}^e_\blacktriangle(k_e)$. A similar convention is used for the associations. This is the abstract description of the main pattern:

$$e_g(k_e : K_e) \triangleq (\, e_{s_1}(k_e) \,|\, \cdots \,|\, e_{s_n}(k_e)\,) \,.\, \mathcal{C}^{e_g}(k_e)$$

$$e_{s_i}(k_e : K_e) \triangleq \mathcal{P}^{e_{s_i}}(k_e) \,.\, (\, \mathcal{M}^{e_{s_i}}_\star(k_e)^* \,|||\, \mathcal{A}^{e_{s_i}}_\star(k_e)\,)$$

$$\mathcal{M}^{e_{s_i}}_\star(k_e : K_e) \triangleq \mathcal{M}^{e_{s_i}}(k_e) \mid \mathcal{M}^{e_{s_i}}_\blacktriangle(k_e)$$

$$\mathcal{A}^{e_{s_i}}_\star(k_e : K_e) \triangleq \mathcal{A}^{e_{s_i}}(k_e) \parallel \mathcal{A}^{e_{s_i}}_\blacktriangle(k_e)$$

To illustrate further this pattern, it is instantiated for the inheritance hierarchy in Fig. 2, with $e_g := e_1$, $n := 2$, $e_{s_1} := e_2$, and $e_{s_2} := e_4$.

$$\begin{aligned}
e_1(k:K) &\triangleq (\, e_2(k) \,|\, e_4(k)\,) \,.\, C^{e_1}(k)\\
e_2(k:K) &\triangleq P^{e_2}(k) \,.\, (\, M^{e_2}_\star(k)^* \,|||\, \Theta(e_1,\cdots,a_1,\cdots)\,)\\
e_4(k:K) &\triangleq P^{e_4}(k) \,.\, (\, M^{e_4}_\star(k)^* \,|||\, A^{e_4}_\star(k)\,)\\
M^{e_2}_\star(k:K) &\triangleq M^{e_2}(k) \mid M^{e_1}(k)\\
M^{e_4}_\star(k:K) &\triangleq M^{e_4}(k) \mid M^{e_3}_\star(k)\\
M^{e_3}_\star(k:K) &\triangleq M^{e_3}(k) \mid M^{e_1}(k)\\
A^{e_4}_\star(k:K) &\triangleq \Theta(e_4,\cdots,a_3,\cdots) \parallel A^{e_3}_\star(k)\\
A^{e_3}_\star(k:K) &\triangleq \Theta(e_3,\cdots,a_2,\cdots) \parallel \Theta(e_1,\cdots,a_1,\cdots)
\end{aligned}$$

We may summarize these definitions as follows. An entity can be created using a producer from either e_2 or e_4. An entity of e_2 can be modified using modifiers of e_1 or e_2. It can be associated through a_1 with some other entities. It is deleted by executing a consumer of e_1. An entity of e_4 can be modified by using any modifier from e_1, e_2, e_4. It can be associated with other entities by using associations a_1, a_2, a_3. It is deleted by executing a consumer of e_1. Note that there is no process definition for e_3, because it is neither a leaf nor the top of the hierarchy.

5.2 Entity attribute function patterns

Attributes are defined by recursive functions. The functions are more varied in style, but a few simple patterns still cover quite a large number of cases. We present two of them.

When an attribute b_k is a key of an entity type e, the structure of its recursive function has the following pattern. For the sake of simplicity, assume that this entity has only one producer and one consumer.

```
$b_k(s) \triangleq$
    match $s$ with
        $\varepsilon$                  $\rightarrow \emptyset$
        $s' \vdash P^e(k_e, \_)$  $\rightarrow \{k_e\} \cup b_k(s')$
        $s' \vdash C^e(k_e)$      $\rightarrow b_k(s') - \{k_e\}$
        $s' \vdash \_$             $\rightarrow b_k(s')$
```

Producers add key values to the set of keys of active entities; consumers remove them. The initial value of a key attribute (i.e., for the empty trace) is the empty set ($\emptyset$), because there are no active entities in the beginning. Other simple attributes b_o (e.g., *ward_name*, *patient_name*) of e have the following pattern:

```
$b_o(s, k_e) \triangleq$
    match $s$ with
        $\varepsilon$                              $\rightarrow \bot$
        $s' \vdash P^e(k_e, \cdots, v_{b_o}, \cdots)$  $\rightarrow v_{b_o}$
        $s' \vdash C^e(k_e)$                  $\rightarrow \bot$
        $s' \vdash M^e(k_e, \cdots, v_{b_o}, \cdots)$  $\rightarrow v_{b_o}$
        $s' \vdash \_$                         $\rightarrow b_o(s', k_e)$
```

The initial value of a simple attribute is provided by a producer and further changed by a modifier. When an entity is deactivated by a consumer, the attribute becomes undefined.

Attributes arising from associations also follow patterns (e.g., *treating_ward* or *treated_patients*).

6 Conclusion

This paper is based upon methods proposed in the early 80's, which have been studied intensively for many years and used successfully in many applications. It provides a new style of black-box specification that allows developers to rigorously describe the interaction of a system with its environment solely from the user's point of view. It combines and adapts various techniques, each exploited for aspects that it treats particularly well, to obtain a notation that is light but complete enough for specifying a large class of ISs.

Using relations instead of functions is a slight generalization of the initial Cleanroom model of black-box specifications. It supports a simple form of nondeterminacy in specifications. Relations between input histories and output histories (i.e., relations between X^+ and Y^+), as proposed in [2] and [32], are even more general. However, for a large class of ISs, relations on $X^+ \times Y$ are sufficient and easier to manage, since it is not necessary to deal with the output history.

Using PEs instead of regular expressions yields a notation that is more expressive in the sense that it deals with complex ordering constraints. The operations chosen among those in CSP and LOTOS have been sufficient to formalize the semantics of processes described informally in the JSD method. The choice of using concatenation instead of prefixing and sequential composition makes the set of inference rules a bit more complex than in LOTOS or CCS, but the specifications shorter.

Despite its mathematical foundations, this new specification method does not require the analyst to have a strong mathematical background. PEs are close to JSD structure diagrams. Furthermore, because entities closely correspond to relations in a relational database, input-output rules can be defined by using SQL statements. This short conceptual distance between this new kind of black-box specification and a corresponding SQL-based implementation should make the design phase straightforward and reduce the learning curve for software designers to apply this new method. Similarly, entity types closely correspond to classes in an objet-oriented approach. Hence, there should be a natural refinement of black-box specifications by an object-oriented design.

Our personal experimentations with EB^3 show that it can conveniently be used for the ISs following the patterns of Sect. 5. Nevertheless, real-world systems can be very complex. We do not know yet how far EB^3 can go in managing this complexity.

6.1 Future work

This work leaves open a number of research issues. The ultimate goal of EB^3 is to automatically generate a fully operational system from a specification. An interpreter has been developed to determine if an input trace is valid. It remains to extend this interpreter to compute recursive functions and input-output rules. The next step will be to increase interpreter efficiency to provide response time competitive with that of a conventional implementation of an IS.

A prototype that generates a simple user interface from an EB^3 specification has been built. This interface is sufficient for specification validation and prototyping, but it needs to be improved in order to become an ergonomic, production-strength interface. Research work in model-based interface development environments (MB-IDE) has explored this issue, but without any integration to a formal specification of the functional requirements [30]. By integrating research results in MB-IDE and formal specifications, the ultimate goal of EB^3 becomes quite realistic for a large class of information systems. Finally, the integration of EB^3 specification with existing systems must also be investigated, because new information systems are typically linked to existing systems.

EB^3 specifications can also be refined into state-based implementations. In [17], a refinement strategy and proof rules are proposed for B machines. This work could be extended to generate a partial implementation of an EB^3 specification into object-oriented languages like C++ or Java. The verification of this implementation could be carried out by automatically generating functional test scenarios from the EB^3 specification and automatically checking implementation conformance. A bridge could also be generated to use verification tools developed for CSP and LOTOS. A functional test scenario is represented by a sequence of input-output pairs. An EB^3 specification can be easily extended to include an operational profile specification, which describes the probability of occurrence of a trace. These probabilities are expressed according to the structure of the process expressions.

Acknowledgements. The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the *Fonds québécois de la recherche sur la nature et les technologies* (FQRNT).

References

1. Amyot, D., Eberlein, A.: An Evaluation of Scenario Notations for Telecommunication Systems Development. In: 9th International Conference on Telecommunications Systems (ICTS'01). Dallas, USA March 2001
2. Broy, M., Dederichs, F., Dendorfer, C., Fuchs, M., Gritzner, T.F., Weber, R.: The Design of Distributed Systems – an Introduction to FOCUS. Technische Universität München, Institut für Informatik, TUM-I9203 1992
3. Boudriga, N., Mili, A., Zalila, R., Mili, F.: A Relational Model for the Specification of Data Types. Computer Languages 17(2): 101–131, 1992

4. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14(1): 25–59, 1987
5. Booch, G., Rumbaugh J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley, Reading, MA 1999
6. Cameron, J.R.: JSP and JSD: The Jackson Approach to Software Development. Second Edition, IEEE Computer Society Press, Washington 1989
7. Chen, P.: The Entity Relationship Model – Towards a Unified View of Data. ACM Transactions on Database Systems 1(1): 9–36, 1976
8. Davis, A.: Requirements Engineering. Prentice Hall, Englewood Cliffs 1992
9. Deck, M.D.: Data Abstraction in the Box Structures Approach, Proc. 3rd Annual Int. Conf. on Cleanroom Software Engineering Practices 1996
10. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 3rd edition, Addison-Wesley 2000
11. Fraikin, B., Frappier, M.: EBSPAI: an Efficient Process Algebra Interpreter. 5th Workshop on Tools for System Design and Verification (FM-TOOLS 2002), Reisensburg Castle, Günzburg, Germany July 15–17 2002
12. Fraikin, B., Frappier, M.: EB^{3}PAI: an Interpreter for the EB^3 Specification Language. 15th International Conference on Software & Systems Engineering & their Applications. Paris, France December 3–5 2002
13. Frappier, M., Mili, A., Desharnais J.: Defining and Detecting Feature Interactions, In: Proc. IFIP TC2 Working Conf. on Algorithmic Languages and Calculi 1997
14. Frappier, M., St-Denis, R.: A Specification Method for Cleanroom's Black Box Description, Proc. 31st Hawaii Int. Conf. on System Sciences 1998
15. Frappier, M., St-Denis, R.: Combining JSD and Cleanroom for Object-Oriented Scenario Specification. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems. Kluwer Academic Publishers, Boston 1999
16. Frappier, M., St-Denis, R.: Specifying a Cleanroom Black Box Using JSD. In: Frappier, M., Habrias, H. (eds.) Software Specification Methods: An Overview Using a Case Study. Springer, London 2000
17. Frappier, M., Laleau, R.: Verifying Event Ordering Properties for Information Systems. The third International Conference of B and Z Users, Lecture Notes in Computer Science, vol. 2651. Springer-Verlag, Turku, Finland June 4–6 2003
18. Frappier, M., Fraikin, B., Laleau, R., Richard, M.: Automatic Production of Information Systems. In: AAAI Symposium on Logic-Based Program Synthesis, Stanford University, Stanford, CA March 25–27 2002
19. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs 1985
20. Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y., Chen, C.: Formal Approach to Scenario Analysis. IEEE Software 11(2): 33–41, 1994
21. Jackson, M.: System Development. Prentice Hall, Englewood Cliffs 1983
22. Jarke, M., Kurki-Suonio, R., Eds.: Special Issue on Scenario Management. IEEE Transactions on Software Engineering 24(12), 1998.
23. Karlsson, E.-A.: An Extension of the Black Box Approach to System Specification, Proc. 3rd Annual Int. Conf. on Cleanroom Software Engineering Practices 1996
24. Linger, R.C.: Cleanroom Process Model. IEEE Software 11(2): 50–58, 1994
25. Lustman, F.: Specifying Transaction-Based Information Systems with Regular Expressions. IEEE Transactions on Software Engineering 20(3): 207–217, 1994
26. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs 1989
27. Mills, H.D., Linger R.C., Hevner, A.R.: Principles of Information Systems Analysis and Design. Academic Press, Orlando, FL 1986
28. Oshana, R.S.: Tailoring Cleanroom for Industrial Use. IEEE Software 15(6): 46–55, 1998
29. Prowell, S.J.: Sequence-Based Software Specification. Ph.D. Dissertation, University of Tennessee 1996
30. Puerta, A.R.: A Model-Based Interface Development Environment. IEEE Software 14(4): 41–47, 1997
31. Sridhar, K.T., Hoare, C.A.R.: JSD Expressed in CSP, Technical Monograph PRG-51, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, July 1985, pp. 334–363. Reprinted in [6]
32. Wang, Y., Parnas, D.L.: Simulating the Behavior of Software Modules by Trace Rewriting. IEEE Transactions on Software Engineering 20(10): 750–759, 1994
33. Weidenhaupt, K., Pohl, K., Jarke, M., Haumer, P.: Scenarios in System Development: Current Practice. IEEE Software 15(2): 34–45, 1998
34. Yeung, W.L.: Denotational Semantics for JSD, In: 4th Asia-Pacific Software Engineering and International Computer Science Conference, IEEE Computer Society Press, December 02–05 1997, pp. 72–80

Marc Frappier is a professor of software engineering at the Université de Sherbrooke. He earned a Ph.D. in computer science from the University of Ottawa in 1995. His research interests include software specification and synthesis, software measurement, and project management. He held several positions in industry prior to his academic career, both at the technical and management levels. He is also an independent industrial consultant.

Richard St-Denis received the B.Sc. and M.Sc. degrees in computer science from the Université de Montréal in 1975 and 1977, respectively, and the Ph.D. degree in applied sciences from École Polytechnique de Montréal in 1992. He is currently a professor of computer science at the Université de Sherbrooke, where his research interests include reactive systems, discrete-event systems, and software engineering. He has published a book in French on programming with the *Sparc* assembly language.