A Specification Method for Cleanroom’s Black Box Description

M. Frappier and R. St-Denis
Département de mathématiques et d’informatique
Université de Sherbrooke, Sherbrooke, Québec, Canada, J1K 2R1
email: {Marc.Frappier,Richard.St-Denis}@dmi.usher.ca

Abstract
We describe a specification method for Cleanroom’s black box description. Our specifications are written using entity structure diagrams drawn from the JSD method. We formalize this concept using operators inspired from process algebras. Entity structure diagrams streamline the specification of valid input sequences and their corresponding output. They may be graphically represented, which enhances their readability. This specification method also reduces the conceptual distance between a black box specification and an object-oriented design, since entities are natural candidates for objects.

1 Introduction
The Cleanroom approach to software development [10] uses black box specifications for describing the observable behavior of a software system. A black box specification is a function between the history of inputs (sequences of inputs) and outputs. It is a very abstract description of a software system: by focusing only on inputs and outputs, and disregarding traditional information like state transitions, it allows one to concentrate on just what a system must do, and not how it does it.

Writing complete, precise, concise, readable black box specifications is not an easy task. Several Cleanroom practitioners use informal, or semi-formal, descriptions of black boxes (e.g., [8]), but they tend be quite long, and their informality may lead to ambiguities. Some formal approaches have been proposed [2, 12], but they may also suffer from incoherence, because of the combinatorial explosion due to the inductive nature of their notation.

In this paper, we explore a new method for writing black box specifications in a precise, concise and readable manner. To achieve precision, we use first-order predicate logic. Our formal language is mainly composed of basic operations on sequences. To achieve conciseness and readability, we import several concepts from the JSD method [3] and process algebras [11], and adapt them to write sequence-based specifications.

The concept of entity structure diagram in JSD allows us to restrict combinatorial explosion. We have found that JSD diagrams can describe in a very concise manner the set of valid input sequences. Moreover, they allow us to extract instances of entities from an input sequence and to easily write axioms to describe the input-output behavior, based on these instances. Since entity structure diagrams in JSD had never been formalized, we use parallel composition operators, inspired from process algebras, to provide them with a formal semantics.

Our paper is structured as follows. In Section 2, we describe the mathematical background of our approach, mainly predicate logic and operations on sequences. In Section 3, we present the specification method. Along its presentation, we use a running example, the library system, to illustrate its concepts. We conclude with some remarks in Section 4.

2 Mathematical Foundations
2.1 Black Box Specifications
Following Davis [5], we distinguish between the specification of static software systems and the specification of dynamic software systems. Static systems simulate a simple input-output behavior, where the output is determined by the current input, while dynamic systems 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. An example of a static system is a compiler, whose output is determined exclusively by the current input; an example of a dynamic system is an operating system, where the same command may elicit different outputs depending
on the history of past commands. One may consider that static systems are a special case of dynamic systems.

The specification of a dynamic system (a black box specification) is defined by the following parameters:

- An input space, say $I$, which represents individual inputs that are submitted to the system. From the input space, we derive the set $H$ of input histories, where an input history is defined as a non-empty finite sequence of elements of the input space.
- An output space, say $O$, which represents the outputs (responses) that the system may return following an input.
- An input-output relation, say $R \subseteq H \times O$, which includes pairs that are considered correct.

For the sake of nondeterminism, the same input history may be associated with more than one output; also, for the sake of abstraction, this model allows us to represent systems that define the behavior of state machines without forcing us to define states. Our definition is an extension of Mills’ et al [10], in that we use a relation instead of a function. Other authors use relations on the input history and the output history [13], which provide a slightly more powerful form of nondeterminism. Although, our form of nondeterminism is sufficiently general for a large class of applications.

As a simple example of a black box specification, we may consider a counter. The initial value of the counter is zero; it is incremented by 1 after submitting input inc; it is decremented by 1 after submitting input dec; the value of the counter is displayed by submitting input val. The input set of the counter object is given by:

$$I \triangleq \{\text{inc, dec, val}\} .$$

The output set of the counter object is given by:

$$O \triangleq \text{Integer} \cup \{\tau\} .$$

The symbol $\tau$ denotes the case where the counter produces no visible output for a given sequence of inputs. We now provide some examples of elements (pairs) of a relation $R$ specifying the counter object. We use a dot (‘.’) for the concatenation of input sequences; we use $s \cdot R \cdot s'$ to denote that the pair $(s, s')$ is an element of relation $R$.

$$\begin{align*}
    \text{val} & \cdot R \cdot 0 \\
    \text{inc} & \cdot R \cdot \tau \\
    \text{inc} \cdot \text{val} & \cdot R \cdot 1 \\
    \text{val} \cdot \text{inc} \cdot \text{dec} & \cdot R \cdot \tau \\
    \text{val} \cdot \text{inc} \cdot \text{dec} \cdot \text{val} & \cdot R \cdot 0 \\
    \text{val} \cdot \text{inc} \cdot \text{dec} \cdot \text{inc} \cdot \text{val} & \cdot R \cdot 1
\end{align*}$$

The first pair indicates that the counter produces an output value of zero after receiving input element val; it satisfies the requirement that the initial value of the counter is zero. The second pair indicates that the counter produces no output after receiving input element inc. The third pair illustrates why we need a relation on input sequences, instead of just considering input elements, to specify the behavior of the counter. For input val, the output of the counter depends on the number of dec and inc that were previously submitted to the counter. For instance, after starting the system, if the user submits input inc, and then submits input val, the output of the counter is 1, according to the third pair. Hence, in an input sequence $u_1 \ldots u_n$, we interpret $u_1$ as the first input submitted since the start of the system, $u_2$ as the second, and $u_n$ as the last input submitted.

### 2.2 Basic Mathematical Notation

To write relations, we consider the language of first order logic with the usual connectives in the following binding order from highest to lowest: $\neg$, $\land$, $\lor$, $\{\ldots, \}, \{\ldots\}$, $\forall$, $\exists$, $\wedge$, $\vee$, $\wedge$, $\vee$. Binding order is modified by means of parentheses. We also use $(\ldots, \ldots)$ to denote a tuple (i.e., an element of a Cartesian product) and $(\ldots)$ to denote a set. We use $\mathbb{P}(X)$ and $\mathbb{P}(X)$ to denote, respectively, the power set and the finite power set (set of all finite subsets) of $X$. We use $\pi_i$ to denote the projection function on the $i$th attribute of a tuple. By extension, $\pi_{i..j}$ denotes the projection on attributes $i..j$. Other function symbols and relation symbols may be used in the context of a particular specification. We use $x, y, z, \ldots$, possibly subscripted, for individual variables denoting sequences over $I$; we use $u, v$ to denote elements of $I$; we use $a, b, c$ for constants. Relational constants (i.e., constant symbols whose denotations are binary relations) are written using capital letters (usually $R$, or a meaningful name like Library). The composition of two functions is denoted by $f \circ g$ and defined as $(f \circ g)(x) \triangleq f(g(x))$. We may extend a function $f$ to a set $X$ in a point-wise manner, i.e., $f(X) \triangleq \{f(x) | x \in X\}$. Let $e(x)$ be a term where variable $x$ may occur; we may extend a commutative, associative binary operation $\Phi$ with unit $e$ by quantifying over a finite set in the following manner.

$$\begin{align*}
    \Phi_{x \in O} e(x) & \triangleq e \\
    \Phi_{x \in X} e(x) & \triangleq e(x) \Phi (\Phi_{y \in X - \{x\}} e(y))
\end{align*}$$
2.3 Sequences

We use several classical operations on sequences. We provide below a list of them with an informal description. For simplicity, we do not distinguish between a one element sequence and its corresponding element.

2.3.1 Basic Operations on Sequences

Symbol $\varepsilon$ denotes the empty sequence. We use $u \cdot x$ to denote the left append of element $u$ to sequence $x$. We use $x \cdot u$ to denote the right append of element $u$ to sequence $x$. We use $x \downarrow A$ to denote the projection of sequence $x$ on alphabet $A$; we use $x \uparrow A$ to denote the projection of sequence $x$ on alphabet $A$; they are defined as follows.

$\varepsilon \downarrow A = \varepsilon$

$u \in A \Rightarrow (u \cdot x) \downarrow A = u \cdot (x \downarrow A)$

$u \notin A \Rightarrow (u \cdot x) \downarrow A = x \downarrow A$

$\varepsilon \uparrow A = \varepsilon$

$u \notin A \Rightarrow (u \cdot x) \uparrow A = u \cdot (x \uparrow A)$

$u \in A \Rightarrow (u \cdot x) \uparrow A = x \uparrow A$

We use $A^+$ to denote the set of all sequences constructed from elements of $A$; we also have $A^* \triangleq A^+ \cup \{\varepsilon\}$. We use $\text{prefix}(x)$ to denote the set of prefixes of sequence $x$, i.e., $\text{prefix}(x) \triangleq \{y \exists z : x = y \cdot z\}$. We use $\text{last}(x)$ to denote the last element of sequence $x$, provided $x$ is not empty. We use $\text{head}(x)$ to denote the first element of sequence $x$, provided $x$ is not empty. We use $\text{tail}(x)$ to denote all but the first element of sequence $x$, provided $x$ is not empty. We use $\text{front}(x)$ to denote all but the last element of sequence $x$, provided $x$ is not empty. We use $\#x$ to denote the length of sequence $x$.

2.3.2 Interleave

The interleave of two sequences $x,y$ is the set of sequences resulting from all the possible ways of merging $x$ and $y$. We may formally define this operator as follows.

$x \| \varepsilon \triangleq \varepsilon \| x \triangleq \{x\}$

$(u \cdot x) \| (v \cdot y) \triangleq u \cdot (x \| (v \cdot y)) \cup v \cdot ((u \cdot x) \| y)$

2.3.3 Synchronized Parallel Product

To represent the concurrent execution of two entities, we need to define a parallel composition operator where the two entities synchronize on common inputs. The synchronized parallel product of two sequences $x$ and $y$, defined over alphabets $\Sigma_1$ and $\Sigma_2$ respectively, is defined as follows.

$\langle x, \Sigma_1 \rangle \| \langle y, \Sigma_2 \rangle \triangleq \{\langle z, \Sigma_3 \rangle \ | \Sigma_3 = \Sigma_1 \cup \Sigma_2 \wedge z \in \Sigma_3 ^* \wedge z \downarrow \Sigma_1 = x \wedge z \downarrow \Sigma_2 = y\}$

2.3.4 Extended Regular Sets

An extended regular set (ERS) is a set of sequences over an alphabet $\Sigma$. It may be constructed using elementary extended regular sets and the following operators, extended point-wise to sets when necessary: $\cdot$, $\|$, $\ast$, $\ast$. Operator $\mid$ is the same as the union of two sets; we use it, instead of $\cup$, because it is the symbol typically used in regular expressions or process algebras; it represents a choice between two sets. We use the following binding order from highest to lowest: $\{\cdot, \|, \ast, \ast\}$.

An elementary ERS may be constructed using variables and the special symbol “$\ast$”. Variables are useful for defining ERS over an infinite alphabet. The special symbol “$\ast$” is useful for creating expressions with “don’t care” values; it reduces the number of variables required to write an expression. Our ERS are typically defined over an alphabet structured as the Cartesian product of elementary sets. We use a tuple-like notation to define elementary ERS. Provided below is an example of tuple with its formal meaning. In this example, let $a$ be a variable ranging over an elementary set and $T$ be an elementary set.

$\langle a, \rangle \triangleq \{u \exists b : b \in T \wedge u = \langle a, b \rangle\}$

3 Specification Method

A black box specification is defined by an input space $I$, an output space $O$, and an input-output relation $R$ which can be viewed as the observable behavior of a mechanism hidden within the box. To specify this kind of black box, we propose a method broken into five steps:

1. definition of input and output spaces $I$ and $O$;
2. definition of entities included in the system and description of their individual input behavior;
3. construction of well-formed input sequences (i.e., those defining the free input behavior of the system) from individual input behaviors;
4. definition of constraints on the well-formed input sequences (input sequences satisfying the constraints express the legal input behavior of the system);
5. definition of relation \( R \) (i.e., outputs from \( O \) in response to sequences of inputs from \( I \)) for a complete definition of the input-output behavior.

All these steps consist in a declarative process based on mathematical tools. They are domain dependent, except the third one which is generic and can be automatically achieved. Before giving more details on each of these steps, let us introduce a running example.

### 3.1 The Library Example

To illustrate the application of our specification method, a small example is offered, a library system borrowed from Cameron [3]. In this system, a library member can borrow a book, renew a loan, or reserve a book only if he has joined but not left the library. He can also cancel any of his book reservations. Finally, a restriction is imposed on the total number of books checked out at one time by a member. Actions performed by a member represent transactions initiated by the library staff. In addition, the library staff can add a book to the library or discard a book previously acquired. A number of reports are also available by executing appropriate commands. This short description defines a first boundary of the system. As usual, some details will be elicited and added during the formal specification of the system.

### 3.2 Specification of Input and Output Sets

The first step of our specification method consists in determining the input and output spaces of the system. The input set \( I \) and output set \( O \) are drawn up from an analysis of the system. Several criteria may be applied to filter inputs and outputs that are to be included in \( I \) and \( O \), respectively.

An input is composed of a label, representing an action or an event of the environment, and a fixed number of attributes. The domain of an attribute is a set of constants that represent its possible values. An output has no predefined format but belongs to a set that determines its type.

Generally, input set \( I \) (output set \( O \)) is defined as the union of some input sets \( I_1, \ldots, I_n \) (output sets \( O_1, \ldots, O_n \)). Each output set \( O_i \) is associated to input set \( I_i \). An output that belongs to \( O_i \) is produced when the last input of an input sequence belongs to \( I_i \).

The following list describes the input sets for the library system.

\[
\begin{align*}
I_1 & \triangleq \{ \text{Acquire} \} \times \text{BookId} \times \text{Title} \times \text{Author} \times \text{Date} \\
I_2 & \triangleq \{ \text{Lend} \} \times \text{BookId} \times \text{MemberId} \times \text{Date} \\
I_3 & \triangleq \{ \text{Renew} \} \times \text{BookId} \times \text{Date} \\
I_4 & \triangleq \{ \text{Reserve} \} \times \text{ReservationId} \times \text{BookId} \times \text{MemberId} \times \text{Date} \\
I_5 & \triangleq \{ \text{Take} \} \times \text{ReservationId} \times \text{Date} \\
I_6 & \triangleq \{ \text{Return} \} \times \text{BookId} \times \text{Date} \\
I_7 & \triangleq \{ \text{Cancel} \} \times \text{ReservationId} \\
I_8 & \triangleq \{ \text{Sell} \} \times \text{BookId} \\
I_9 & \triangleq \{ \text{Discard} \} \times \text{BookId} \\
I_{10} & \triangleq \{ \text{Join} \} \times \text{MemberId} \times \text{Name} \times \text{Phone} \times \text{LoanLimit} \\
I_{11} & \triangleq \{ \text{ModifyMember} \} \times \text{MemberId} \times \text{Phone} \times \text{LoanLimit} \\
I_{12} & \triangleq \{ \text{Leave} \} \times \text{MemberId} \\
I_{13} & \{ \text{ListBookDescr} \} \times \text{BookId} \\
I_{14} & \{ \text{ListOverdueBooks} \} \times \text{Date} \\
I_{15} & \{ \text{ListNewBooks} \} \times \text{Date} \\
I_{16} & \{ \text{ListReservationsForOverdueBooks} \} \times \text{Date} \\
\end{align*}
\]

The following list describes the output sets of the library. We only list output sets of inputs producing a visible output. Thus, we have \( O_1 \) to \( O_5 \) and \( O_7 \) to \( O_{12} \) which are defined as \( \{ ? \} \).

\[
\begin{align*}
O_1 & \triangleq \text{BookId} \times \text{Title} \times \text{Date} \times \text{MemberId} \\
O_{16} & \triangleq \text{BookId} \times \text{Title} \times \text{Author} \times \{ "$\text{Book not found}" \} \\
O_{17} & \triangleq \{ \text{MemberId} \times \text{BookId} \} \\
O_{18} & \{ \text{Title} \times \text{Author} \times \text{BookId} \} \\
O_{19} & \{ \text{MemberId} \times \text{Name} \times \text{Phone} \times \text{BookId} \times \text{Title} \} \\
\end{align*}
\]

As example, consider an input belonging to \( I_6 \). It has Return as a label and two attributes: a book id and a return date. For instance, if \( u = \langle \text{Return, TK1, 23/09/98} \rangle \), then \( u \in I_6 \). The corresponding output is formed from the id of the returned book, its title, the return date, and the id of the member who returned the book. The input set \( I \) and output set \( O \) for the library system are defined as the union of all these input sets and output sets, respectively, as follows.

\[
I \triangleq \bigcup_{i=1}^{16} I_i \quad O \triangleq \bigcup_{i=1}^{16} O_i
\]

### 3.3 Definition of Entities

The second step of our specification method is largely inspired from the entity structure step of the JSD method [7]. We first identify the entities composing the system. Then, we describe the behavior of each entity in terms of input histories. The main objective of this step is to introduce some ordering constraints on the inputs identified in the first step. Jackson proposed to express these constraints by using entity structure diagrams which are essentially a diagrammatic notation for regular expressions. Each diagram describes the inputs of one entity. There are as many instances
of a diagram as there are instances of the corresponding entity. Since the same input may belong to different structure diagrams, the ordering constraint on an input is the intersection of constraints imposed by the structure diagrams in which it appears. In fact, the JSD method implicitly assumes a parallel composition between entities with synchronization on common inputs. We define the behavior of an entity from extended regular sets, which we may graphically represent.

Let \( e_1, \ldots, e_n \) be the entities of the system. Let \( E = \{ e_1, \ldots, e_n \} \). In our method, each entity can be interpreted as a type of object. The behavior of an entity \( e \in E \) is defined using an alphabet \( \Sigma_e \subseteq I \). To designate a particular object of type \( e \), we associate a set of keys \( K_e \) to \( e \). Typically, \( K_e \) is a Cartesian product of attribute domains shared by inputs that belong to \( \Sigma_e \). Let \( f_e : K_e \rightarrow \mathbb{P}(\Sigma_e^*) \) be a total function that provides a set of input sequences for a key \( k \). A sequence of \( f_e(k) \) is called an instance of entity \( e \) with key \( k \). The function \( f_e \) is typically defined using the prefixes of sequences defined by an extended regular set over \( \Sigma_e \). We will use \( f_e \) in Section 3.4 to form input sequences by synchronizing the entities. The distinction between an instance and an object is rather subtle: an instance represents one of the possible behaviors of an object. For example, a particular book may be acquired and discarded, or acquired, lent, returned, and discarded. These two possible behaviors are two instances of the same object.

The library has four entities: \( \text{Library} \triangleq \{ \text{book}, \text{member}, \text{reservation}, \text{query} \} \). The behavior of these entities could be described by a specification that includes four entity structure diagrams, one per each entity. As an example, Figures 1 and 2 show two diagrams which are similar to Jackson's entity structure diagrams except that they are syntax trees for the underlying expressions defining the extended regular sets. In these figures, “*”, “\(|\)”, and “\( ||| \)” in the top right corner of boxes denote iteration, selection, and interleave, respectively. The absence of an operator denotes a sequence (i.e., “\( = \)”).

The alphabet of entity book is \( \Sigma_{\text{book}} \triangleq I_1 \cup I_2 \cup I_3 \cup I_4 \cup I_5 \cup I_6 \cup I_7 \cup I_8 \cup I_9 \) and its set of keys is \( \text{Book.Id} \). The function \( f_{\text{book}} \) is defined as follows: \( f_{\text{book}}(b, \text{id}) \) provides a set of prefixes of any sequence generated from the extended regular set \( book(b, \text{id}) \).

The extended regular set for the entity book is the formal counterpart of the diagram in Figure 1. The term \( f_{\text{book}}(b, \text{id}) \) provides a set of prefixes of any sequence generated from the extended regular set \( book(b, \text{id}) \). The function \( book \) is defined using auxiliary functions \( active \_book \) and \( loan \). These functions will be used later in the specification of the input-output behavior. A
Given a set of entities $E$, $e \in E$, and $k \in K_e$, $f_e(k)$ yields the set of all possible input sequences of the object of type $e$ having key $k$. Therefore, $f_e(k)$ includes all instances of only one object. From this set, we construct the set of instances of entity $e$ (i.e., all objects of type $e$), denoted by $F_e$, and the set of instances of system $E$ (i.e., all objects no matter their type), denoted by $FE$, as follows.

$$F_e \triangleq \bigcup_{k \in K_e} f_e(k) \quad F_E \triangleq \bigcup_{e \in E} F_e$$

From the individual behaviors, we define the set of all possible input sequences of the system by using parallel composition operators. This set is called the set of well-formed input sequences. Let $S_E$ denote this set. Well-formed input sequences are constructed by 1) interleaving some instances of objects of same type (to obtain a set $S_e$) and 2) synchronizing instances of objects of different types (to obtain $S_E$). Furthermore, a well-formed input sequence must include no more than one instance of a particular object. Let $G_e$ be the set of functions $g$ that associate one and only one instance of entity $e$ for each key $k$ of a finite subset of $K_e$.

$$G_e \triangleq \{ g | g : \kappa_e \rightarrow \mathbb{Z}^* \land \kappa_e \neq \emptyset \land \forall \kappa_e \in \mathbb{Z}(K_e) \land \forall k : k \in \kappa_e \Rightarrow g(k) \in f_e(k) \}$$

Then, the sets $S_e$ and $S_E$ are defined as follows.

$$S_e \triangleq \{ x | \exists g : g \in G_e \land x \in \bigcup_{k \in \text{dom}(g)} g(k) \}$$
$$S_E \triangleq \{ x | \exists y_1, \ldots, y_n : y_i : y_i \in S_e_1 \land \ldots \land y_n \in S_e_n \land x \in \bigcup_{i=1}^n \{ y_i, \Sigma_e \} - \{ e \} \}$$

From a well-formed input sequence, we must be able to extract instances of objects involved in this sequence. The extraction of instances allows one to reason more easily on input sequences and define constraints and input-output relations in a simple manner. It is formulated in terms of predicates based on a function $\text{sync} \in \mathbb{Z}(F_E) \rightarrow \mathbb{Z}(S_E)$ that returns the parallel composition of a set of instances.

$$\text{sync}(I) \triangleq \{ x | \exists g_1, \ldots, g_n : g_1 \in G_{e_1} \land \ldots \land g_n \in G_{e_n} \land y_1 \in \bigcup_{k \in \text{dom}(g_1)} g_1(k) \land \ldots \land y_n \in \bigcup_{k \in \text{dom}(g_n)} g_n(k) \land x \in \bigcup_{i=1}^n \{ y_i, \Sigma_e \} \land I = \bigcup_{i=1}^n \bigcup_{k \in \text{dom}(g_i)} g_i(k) \}$$

According to the previous definition, one can determine if an instance $x$ is a subsequence of an input sequence $y$ by verifying that the following predicate holds for $x$ and $y$.

$$\text{is_entity_subseq}(x, y) \iff \exists I : x \in I \land y \in \text{sync}(I)$$
This extraction mechanism is enriched with two additional arguments that specify the type of a function $f$ such that $\text{dom}(f) \subseteq K_e$ and a key $k \in K_e$ for a given $e \in E$.

$$\begin{align*}
\text{instance}_f(x, y, f) & \iff \\
\text{is_entity}_f(x, y) \land x \in \bigcup_{k \in \text{dom}(f)} f(k) & \iff \\
\text{inst}_f(k, y, f, k) & \iff \\
\text{is_entity}_f(x, y) \land x \in f(k) & \iff \\
\end{align*}$$

To illustrate our formal definitions, we provide below some examples drawn from the library system for which $K_{\text{book}} = \{\text{QA1, TK1}\}$ and $K_{\text{member}} = \{\text{m1, m2}\}$. Notice that, for the sake of concision, uninteresting attributes have been replaced by “_”.

$$\begin{align*}
\text{book}(\text{m1}) & \triangleq \\
\{x : (\text{Acquire, m1}, \ldots, \ldots) \cdot (\text{Sell, TK1}), \\
(\text{Acquire, m1}, \ldots, \ldots) \cdot (\text{Discard, m1}), \\
(\text{Join, m1}, \ldots, \ldots) \cdot (\text{Lend, m1, m1}), \\
(\text{Join, m1}, \ldots, \ldots) \cdot (\text{Return, m1}, \ldots), \\
\ldots \} & \text{book}(\text{TK1}) & \triangleq \\
\{x : (\text{Acquire, TK1}, \ldots, \ldots) \cdot (\text{Sell, TK1}), \\
(\text{Acquire, TK1}, \ldots, \ldots) \cdot (\text{Discard, TK1}), \\
(\text{Join, TK1}, \ldots, \ldots) \cdot (\text{Lend, TK1, m1}), \\
(\text{Join, TK1}, \ldots, \ldots) \cdot (\text{Return, TK1}, \ldots), \\
\ldots \} & \text{book}(\text{m2}) & \triangleq \\
\{x : (\text{Join, m2}, \ldots, \ldots) \cdot (\text{Lend, m2}), \\
(\text{Join, m2}, \ldots, \ldots) \cdot (\text{Discard, m2}), \\
(\text{Join, m2}, \ldots, \ldots) \cdot (\text{Return, m2}), \\
(\text{Join, m2}, \ldots, \ldots) \cdot (\text{Leave, m2}), \\
(\text{Join, m2}, \ldots, \ldots) \cdot (\text{Leave, m2}), \\
(\text{Join, m2}, \ldots, \ldots) \cdot (\text{Leave, m2}), \\
(\text{Join, m2}, \ldots, \ldots) \cdot (\text{Leave, m2}), \\
\ldots \}
\end{align*}$$

In particular, the predicates

$$\begin{align*}
\text{instance}_f(x, y, \text{prefix } \circ \text{member}) & \text{ and } \\
\text{inst}_f(k, y, \text{prefix } \circ \text{member}, m) & \text{ hold if } \\
\begin{align*}
x & \triangleq (\text{Join, m1}, \ldots, \ldots) \cdot (\text{Lend, TK1, m1}, \ldots) \\
y & \triangleq (\text{Acquire, TK1}, \ldots, \ldots) \cdot (\text{Acquire, QA1}, \ldots, \ldots) \cdot \\
& \quad (\text{Join, m1}, \ldots, \ldots) \cdot (\text{Join, m2}, \ldots, \ldots) \cdot \\
& \quad (\text{Lend, QA1, m2}, \ldots) \cdot (\text{Lend, TK1, m1}, \ldots)
\end{align*}
\end{align*}$$

Note that most typical constraints encountered in information systems are already defined in entity structure diagrams. For instance, the following input sequence is not well-formed, and it represents the case of an invalid input submitted to the system: in this case, member m1 has not joined the library yet, but he is trying to borrow a book: the output of the system for that input sequence would typically be an informative error message (see Section 3.6.5).

$$\begin{align*}
& (\text{Acquire, TK1}, \ldots, \ldots) \cdot (\text{Join, m2}, \ldots, \ldots) \cdot \\
& \quad (\text{Lend, TK1, m1}, \ldots)
\end{align*}$$

### 3.5 Specification of Constraints

The set $S_E$ includes all well-formed input sequences. Some of these sequences may be, however, invalid because of constraints prescribed by the environment or imposed on the system. The fourth step of our specification method provides the opportunity to define such constraints. Let $\text{Valid}_E$ be the set of valid input sequences defined as follows.

$$\text{Valid}_E \triangleq \{x \mid x \in S_E \land \text{sat}_E(x)\}$$

Each predicate $p_i$ ($1 \leq i \leq n$) represents a constraint. A constraint is generally defined using Boolean functions on $S_E$. For example, in the library system, there are two constraints. The first constraint restricts a member to borrow no more books than its loan limit. The second constraint ensures that a member cannot leave the library if he has open reservations or books.

To specify these constraints, we use the operator $\downarrow A$ to denote the projection on the label of input elements. We may define this operator and its dual as follows.

$$\begin{align*}
& x \downarrow A \triangleq \{u \mid u \in x \land p_1(u) \land \ldots \land p_n(u)\} \\
& x \uparrow A \triangleq \{u \mid u \in x \land p_1(u) \land \ldots \land p_n(u)\}
\end{align*}$$

We may now specify the constraints as follows.
sat_lend_limit(x) ⇔
∀m : instance_of(m, x, prefix o member) ⇒
check_limit(tail(m), Loan Limit(head(m)), 0) = true

sat_idle_member(x) ⇔
∀m : instance_of(m, x, member) ⇒
#(m \downarrow\{Lend, Take\}) - #(m \downarrow\{Return\}) = 0 ∧
#(m \downarrow\{Reserve\}) - #(m \downarrow\{Take, Cancel\}) = 0

The recursive Boolean function check_limit is defined in
Figure 3. It verifies that a member never borrows
more books than its loan limit during its lifetime. It
has three arguments. The first argument is a sequence
x that is processed from left to right. The second argu-
ment is the current loan limit extracted from inputs
Join and Modify Member. Finally, the third argument
represents the current number of loans. Note that we
use, as a convention, the name of the set defining the
domain of an attribute (e.g., Loan Limit) as a projec-
tion function (i.e., Loan Limit) that returns the value
of the attribute for a given input.

check_limit(x, l, m) ≡
if x = ε then
  return true
else
  let u = head(x) in
  if u ∈ {Lend, \ldots} \cup \{Take, \ldots\} then
    if n + 1 > l then
      return false
    else
      return check_limit(tail(x), l, n + 1)
  else if u ∈ \{Return, \ldots\} then
    return check_limit(tail(x), l, n - 1)
  else if u ∈ \{Modify, Member, \ldots\} then
    return check_limit(tail(x), Loan Limit(u), n)
  else
    return check_limit(tail(x), l, n)

Figure 3: The function check_limit

3.6 Specification of Input-Output Behavior

In the last step of our specification method, the input-
output behavior is defined by providing a number of
axioms or rules which apply to valid input sequences
of the form x-u, where u ∈ I x is the last input on which
emphasis is placed. A rule is written in a predicate
language and has the following form (o ∈ O x).

x-u ∈ ValidLibrary ∧ conjunction ⇒ x-u \in Library \or

The formulation of a rule is done by following some
basic principles. First, the fields of output o are calcu-
lated using, as much as possible, local information (i.e.,
information that appears in input u or near to u). Sec-
ond, when the first principle is inapplicable, the fields
of output o are calculated from attributes included in
the constructors of objects involved in the compu-
tation of o. Such constructors are represented by inputs
that appear in the head of instances. Third, based on
constraints imposed by the structure diagrams or ex-
tended regular sets, irrelevant inputs do not have to be
considered. Then, emphasis is placed on subsequences
suitable to the calculation of o by using appropriate
abstraction mechanisms (e.g., projection, last). Finally,
a rule can refer to many related instances of different
entities. A relationship between entities occurs whenever
these entities have common inputs. Generally, the way
one passes from one instance of entity to another is
done through a common input that owns the keys of
related entities (e.g., Lend, Reserve). The next exam-
pies show the application of these principles and refer
to the library system. In the sequel, for the sake of
concision, we omit the conjunct x-u ∈ ValidLibrary in
each rule.

3.6.1 List Book Description

Let us start with a simple example: the description of
a given book. There are two cases: the case where the
book exists and the case where the book does not exist.
This request can be expressed as follows.

inst_of_key(b, x, prefix o active_book, b id) ⇒
x-{List_Book_Descr, b id}<Library>
  {b id, Title(head(b)), Author(head(b))}

¬\exists b : inst_of_key(b, x, prefix o active_book, b id)
⇒ x-{List_Book_Descr, b id}<Library>“Book not found”

Given an input sequence x-{List_Book_Descr, b id} ∈
ValidLibrary, these rules check if an instance of active
books having key b id appears in sequence x. If it is
the case, then the output is formed from the id, title,
and author of the given book; otherwise, the output is
the string “Book not found”.

3.6.2 Receipt when a Member Returns a Book

The second example is more complex. It describes
the production of a receipt when a book is re-
turned by a member. Since the input sequence is
x-{Return, b id, d}, then there exists a Lend or a Take
before this Return, but possibly with many Renew
between these two inputs (see Figure 1). In the rule be-
low, instance b of active books having key b id is ex-
tracted from sequence x; variable v represents the last
Lend or Take of this instance. We use function Label
to extract the label of an input. Values of the first and
third fields of the output are then obtained directly
from the last input \langle Return, b, j, d, d \rangle. The values of the two other fields are extracted from inputs that precede \langle Return, b, j, d, d \rangle in the sequence according to the value of v and by applying functions Title and Member_Id. In particular, if the loan results from a reservation, then the instance of entity reservation having the key that appears in the input \langle Take, _, _ \rangle is extracted from x to get the id of the member who returned the book.

\[
\begin{align*}
\text{inst}_\text{of key}(b, x, \text{prefix } \circ \text{active_book}, b, j, d) & \land \\
v = \text{last}(b \Downarrow \{\text{Lend, Take}\}) & \land \\
(\text{Label}(v) = \text{Lend } \land \\
o = \langle b, j, d, \text{Title}(\text{head}(b)), d, \text{Member_Id}(v) \rangle) & \lor \\
\text{Label}(v) = \text{Take } \land \\
o = \langle b, j, d, \text{Title}(\text{head}(b)), d, \text{Member_Id}(\text{head}(r)) \rangle & \land \\
\text{inst}_\text{of key}(r, x, \text{reservation }, \text{Reservation_Id}(v)) & \land
\end{align*}
\]

\[\Rightarrow x \cdot \langle \text{Return, b, j, d} \rangle \leftarrow \text{Library} \rightarrow \text{out}\]

### 3.6.3 List of New Books

The next example shows how to produce a list as an output from some selection criteria applied to instances of active books. A list is represented by a sequence constructed from elements of a set and sorted according to a given attribute. We express the sort criteria by using predicate sorted(out, π₁, O) which we may define informally as follows: it states that out is a list of all elements of O sorted on the i-th attribute given by projection function π₁. The following rule is the formalization of a request that gives the list of new books, sorted by title, since a given date d.

\[\begin{align*}
O = \{o \exists b : & \text{instance}_\text{of (b, x, prefix } \circ \text{active_book}) \land \\
& \text{Date}(\text{head}(b)) \geq d \land \\
o = \langle \text{Title}(\text{head}(b)), \text{Author}(\text{head}(b)), \\
& \text{Book_Id}(\text{head}(b)) \rangle \land \\
\text{sorted}(\text{out}, \pi_1, O) & \land
\end{align*}\]

\[\Rightarrow x \cdot \langle \text{List New Books, d} \rangle \leftarrow \text{Library} \rightarrow \text{out}\]

### 3.6.4 List Reservations for Overdue Books

The following rule corresponds to a request that finds the list of reservations for overdue books. It results from the application of the same ideas used in the formulation of rules 3.6.2 and 3.6.3. The symbol Max_Loan_Duration, that appears in this rule, denotes a predefined constant duration beyond which a loan is expired.

\[\begin{align*}
O = \{o \exists r : & \text{instance}_\text{of (r, x, active_reservation }) \land \\
& \exists b : \text{inst}_\text{of key}(b, x, \text{prefix } \circ \text{active_book}, \\
& \text{Book_Id}(\text{head}(r))) \land \\
v_1 = \text{last}(b \Updownarrow \{\text{Reserve}\}) \land \\
\text{Label}(v_1) \in \{\text{Lend, Take, Renew}\} \land
\end{align*}\]

\[d - \text{Date}(v_1) > \text{Max Loan Duration } \land \\
\exists m : \text{inst}_\text{of key}(m, x, \text{prefix } \circ \text{active_member}, \\
\text{Member_Id}(\text{head}(r))) \land \\
v_2 = \text{last}(m \Updownarrow \{\text{Join, Modify Member}\}) \land \\
o = \langle \text{Member_Id}(\text{head}(m)), \text{Name}(\text{head}(m)), \\
\text{Phone}(v_2), \text{Book_Id}(\text{head}(b)), \\
\text{Title}(\text{head}(b)) \rangle \land \\
\text{sorted}(\text{out}, \pi_2, O) \land
\Rightarrow x \cdot \langle \text{List Reservations for Overdue Books, d} \rangle \leftarrow \text{Library} \rightarrow \text{out}\]

### 3.6.5 Specifying Errors

So far, we have specified the input-output behavior for valid input sequences. A complete specification must also define the behavior for sequences containing invalid inputs. For instance, input element \langle Lend, TK1, m1, _ \rangle is invalid in the sequence below; hence, this sequence is not an element of ValidLibrary.

\[\langle \text{Acquire, TK1, _ , m , _} \rangle \cdot \langle \text{Join, m2, _ , _} \rangle \cdot \\
\langle \text{Lend, TK1, m1, _} \rangle \cdot \langle \text{Lend, TK1, m2, _} \rangle \cdot \langle \text{Return, TK1, 23/09/98} \rangle\]

A typical information system would, however, produce an error message when this invalid input is received, and later accept the subsequent input elements as if they were not preceded by invalid input elements. For instance, the output for the sequence above should be \langle TK1, “Book Title”, 23/09/98, m2 \rangle.

This behavior can be specified in a generic manner by defining a robust specification. The function robust defined below takes a specification dealing only with valid input sequences and extends it to cater for input sequences containing invalid input elements.

\[\text{robust}(R) \triangleq \}
\{ (x, u) | (\text{red front}(x), R) \cdot \text{last}(x, u) \in R \lor \\
\text{red front}(x), R) \cdot \text{last}(x) \notin \text{dom}(R) \land \\
u = \text{error} \}
\]

where \text{red}(x, R) is defined as follows

\[
\begin{align*}
\text{if } x = \varepsilon & \text{ then } \\
\text{red}(x, R) = \varepsilon & \\
\text{else if } x \in \text{dom}(R) & \text{ then } \\
\text{red}(x, R) = x & \\
\text{else if } \text{red front}(x), R) \cdot \text{last}(x) \in \text{dom}(R) & \text{ then } \\
\text{red}(x, R) = \text{red front}(x), R) \cdot \text{last}(x) & \\
\text{else } \text{red}(x, R) = \text{red front}(x), R) &
\end{align*}
\]

### 4 Conclusion

We have presented a formal specification method for Cleanroom’s black box description. The crux of our approach is the notion of entity structure diagram drawn...
from the JSD method. This notion allows us to concisely specify the valid input sequences of a system: it is a well-known problem with black box descriptions (and several other specification approaches) that 80% of the specification effort is typically spent on recognizing "invalid" input sequences and defining the corresponding behavior. We address this problem in two phases: first by specifying well-formed sequences using synchronization between entities, and second by applying further constraints on well-formed input sequences to deal with more complex validation requirements. It has the advantage of distributing the complexity over two steps, making it easier to read and understand the specification.

Entity structure diagrams also allow us to extract instances from an input sequence and to define the input-output behavior using axioms based on these instances. Because entities, typically, closely correspond to relations in a relational database, our axioms are quite similar to SQL requests. This short conceptual distance between our black box specification and a corresponding SQL-based implementation should make the design phase easier, and reduce the learning curve for software designers to apply our approach. Similarly, entities also closely correspond to classes in an object-oriented approach; hence, there should be a natural refinement of our black box specifications into an object-oriented design.

Our work leaves open a number of research issues. First, our specifications being formal, it should be possible to execute them by translating them into a logic program in a way very similar to the approach illustrated in [9]. It would allow one to validate a specification or to prove properties about it. The methodic enumeration process of [12] would be another source of inspiration for validation. Second, we would like to investigate refinement rules and verification conditions for implementing a black box specification into an executable system. The traditional approach in Cleanroom is to refine a black box into state boxes and clear boxes, which are in turn recursively refined into other black box descriptions. Another approach which we would like to investigate is to refine our black box specifications into model-based specifications like Z [14] or B [1] (see [6] for a discussion on the issue of choosing between black box and model-based specifications). Finally, the derivation of test sequences from entity diagrams seems quite natural; inspiration for this work should come from the large body of work on testing in algebraic specifications like LOTOS [4].

Acknowledgments The authors would like to thank the anonymous referees for their careful reviews and suggestions for improvement.

References