Formal Refinement of Business Processes

Claudia M. A. da Cunha
Universidade Federal de Pernambuco - Departamento de Informática
50732-970 Recife-PE, Brazil - email: cmac@di.ufpe.br

Abstract
The Information Engineering and the Information Engineering Facility tool have been successfully used over the past years as an efficient way of developing Information Systems on a database environment. At present, code is automatically generated from a procedural specification (written using the Process Action Diagram—PAD) and JMA is currently extending the tool, in order to have the PAD automatically generated from a declarative specification. The main purpose of this work is to make the PAD generation a stepwise refinement task, with the use of Morgan’s Refinement Calculus.

1 Introduction
Over the past few decades, systems developers have felt the increasing need for methods that can provide some support throughout the process of construction of systems. This need stimulated the creation of structured methods for analysis and design, each providing some way of representing the different phases of systems development, often using some kind of diagrammatic technique. However, these methods do not usually offer a way of ensuring that programs are correctly generated and meet their specifications. Because information systems are being increasingly used for automating everyday business decisions, programs for implementing business processes need to be correct with respect to some given specification.

Information Engineering (IE) [3] is a structured method for analysis and design; IE allows the data model to be created making use of the Entity-Relationship diagrammatic techniques [1]; the business processes are specified in a SQL-like language called Process Action Diagram (PAD). For a more detailed information about IE and PAD we refer the reader to [3].

Information Engineering Facility (IEF) [3] is a case tool that combines the IE method with a supporting set of software tools from Texas Instruments. A business process can be specified as a PAD which can then be automatically converted into code. Furthermore, the analyst is able to choose his/her target machine from a pre-determined set of languages, operating systems, etc. Research has been conducted within James Martin Associates (JMA, a Texas Instruments company) in order to have the PAD automatically generated from a declarative specification. Because this generation is automatic, the user has no control over the process and the code obtained is not always the most efficient. Recently, some research has been done into an extension of IE and IEF; JMA is trying to raise the level of the specification language, so that it can become more declarative and less procedural. JMA is currently designing and prototyping such a language and, at the moment, specifications written in it can be automatically converted into PADS.

Although this extension of the IEF gives the analyst more freedom when defining the process, it does not allow him to interfere with the automatic transformations that generate the PAD; once a certain specification is given, a corresponding PAD will be produced, with no intermediate interactions.

Morgan’s Refinement Calculus, as described in [5], is a programming method that takes a specification—that is usually not directly executable—of what the computer is to do, and aims to produce a program which will cause the computer to do it. This method actually calculates the program, as the code is derived from a sequence of refinement steps, where each step is based on a so-called refinement law.

A business process is a defined business activity whose execution may be identified in terms of the input and output of entities of specific types.
This paper proposes the use of this Refinement Calculus for the stepwise transformation of certain kinds of business process specifications into programs. A major benefit for the systems designer is the freedom to change the way in which the code is generated, so that different design decisions can be investigated. Furthermore, the final code will be correct by construction and will meet its specification.

In order to produce derivations of business processes, first we need to select a subset of the laws of the Refinement Calculus to be used in our application, and then to specify business processes using a mathematical notation, so that the refinement laws can be directly applied; new laws are proved from simpler laws of the refinement calculus;

In Section 2 the Refinement Calculus is described and a subset of its laws—the ones relevant to our application—is presented. In Section 3, the mathematical model for expressing business processes is introduced; refinement laws for accessing the database are stated and proved from simpler laws of the Refinement Calculus. The next section presents an example of a business process having its code derived from the specification, using the refinement laws. Section 5 presents the conclusion of our work, highlighting its main points and suggesting some future work.

2 Refinement Calculus

The notion of refinement [7] is all about improving programs: a refinement \( T \) of a specification \( S \) is somehow better—either in terms of efficiency, accuracy, or whatever. We say that

\[ S \subseteq T \]

to express that \( S \) is refined by \( T \).

The Refinement Calculus, as described in [5], shows a way of deriving code from some specification, through the application of convenient laws of refinement. The final code is found at the end of a sequence of development steps, and the validity of each step rests on a refinement law that has been applied.

In the Refinement Calculus, the specifications of processes are given in terms of their pre- and post-conditions and a frame: if the pre-condition is satisfied, the process terminates in a state satisfying the post-condition, having affected (or not) only the variables listed in the frame. Thus, the specification \( x : [\text{true}, \; x = x^2_0] \) has pre-condition \( \text{true} \) (the process terminates), post-condition \( x = x^2_0 \) (after the execution, the variable \( x \) will be equal to the square of its initial value—\( x_0 \)) and frame \( x \) (only \( x \) can be possibly altered in the process).

In order to refine the specifications, we have to choose the appropriate refinement laws to be applied. Here we show some of these laws; more detailed information can be found in [5].

**Law for introducing local block (varI)** If \( w \) and \( x \) are disjoints, then

\[ w : [\text{pre}, \; \text{post}] \subseteq [\text{var} \; x : T; \; \text{and I} \cdot \; w, x : [\text{pre}, \; \text{post}]]. \]

Note that the invariant \( I \) used in the law above is optional and considered to be true if omitted.

**Law for introducing logical constant (conI)** If \( \text{pre} \Rightarrow (\exists \; c : T \cdot \text{pre'}) \), and \( c \) is a fresh name (not occurring in \( w, \; \text{pre}, \; \text{post} \)), and \( w \) and \( c \) are disjoints, then

\[ w : [\text{pre}, \; \text{post}] \subseteq [\text{var} \; c : T \cdot \; w, \; c : [\text{pre'}, \; \text{post}]]. \]

**Law for assignment (assI)** If the pre-condition implies the post-condition with an expression \( e \) replacing the occurrences of a variable \( w \), then the refinement "\( e \) is assigned to \( w \)" is possible; more formally,

\[
\text{if } \text{pre} \Rightarrow \text{post}[w \setminus e], \text{ then } w, z : [\text{pre}, \; \text{post}] \subseteq w := e.
\]
Law for leading assignment For any expression $e$,

$$w, x : [\text{pre}[x \setminus e] \text{ post}[x_0 \setminus e_0]] \subseteq w := e; w, x : [\text{pre} \text{ post}].$$

The expression $e_0$ above abbreviates $e[w, x \setminus w_0, x_0]$.

In the example shown later, the law above is used as “leading (generalised) assignment”, as we will be dealing with assignments of arbitrary elements of a set; the generalised assignment is defined as (from [6]):

If “$<$” is a binary relation symbol, then $w : < e$ for some expression $e$, abbreviates

$$w : [w < e[w \setminus w_0]]$$

For example, $e_l : \in \text{ set}$ abbreviates $el : [\text{set} \neq \emptyset, el \in \text{ set}]$.

Law for contracting the frame (contract frame)

$$w, x : [\text{pre}, \text{ post}] \subseteq w : [\text{pre}, \text{ post}[x_0 \setminus x]]$$

Law for strengthening post-condition (sp)

if $\text{pre}[w \setminus w_0] \land \text{post}' \Rightarrow \text{post}$, then

$$w, x : [\text{pre}, \text{ post}] \subseteq w : [\text{pre}, \text{ post}']$$

Law for iteration (doI) Let $I$—representing the invariant—be any formula; let $V$—the variant—be any integer-valued expression. Then

$$w : [I, I \land \neg(\bigvee i \cdot G_i)]$$

$$\subseteq \text{ do}$$

$$(\Box i \cdot G_i \Rightarrow w : [I \land G_i, I \land (0 \leq V < V_0)])$$

$$\text{ od}$$

Law for skip command

If $\text{pre} \Rightarrow \text{post}$, then $w : [\text{pre} \text{ post}] \subseteq \text{skip}$

Law for following assignment (fassI) For any term $e$,

$$w, x : [\text{pre} \text{ post}]$$

$$\subseteq w, x : [\text{pre} \text{ post}[x \setminus e]];$$

$$x := e$$

3 Applying the Refinement Calculus to IE

Our aim here is to develop a framework which will allow us to use the Refinement Calculus to derive programs from specifications describing business processes. We concentrate on specifications including (conditional) updates on databases, as this is the most interesting problem to be tackled. Inclusion and deletion can be treated similarly. Currently, a business process is specified in a declarative language; the equivalent code generated for it is a PAD.

In order to use the Refinement Calculus, it is necessary to have the declarative specifications of business processes translated into some specification written using the Predicate Calculus. We also need to translate the constructs of the PAD, representing our final code, into (an extension of) Dijkstra’s guarded command language. Furthermore, we introduce new refinement laws to describe access to the database. These new laws are proved from more basic laws of the Refinement Calculus.
3.1 Mathematical Model for Business Processes

To specify the business processes, first we need to model databases in mathematical terms; although our updates to a database are on an entire entity, usually we change only the value(s) of specific attribute(s) within it. Thus, the attribute will be our unit of information of the database.

We give a simple mathematical model to specify a database: basically, the idea is to represent each attribute of each entity type as a map (finite partial function) from identifiers to the corresponding value for that attribute. For each entity type $E$, we declare a constant—$KnownE$—which represents the current set of identifiers allocated for that entity type. Then, for each attribute of $E$, we declare a variable $att$, whose domain is $KnownE$. This records the fact that we perform only updates (but no inclusions or deletions). The identifiers and the values for the attributes are given sets:

\[
\text{[Id, Value]}
\]
\[
\text{var att : Id $\leftrightarrow$ Value}
\]
\[
\text{con KnownE : } \forall \text{ Id}
\]
\[
\text{and dom (att) } = \text{ KnownE}
\]

A major advantage of representing the database in terms of its attributes—rather than in terms of its entities—is that the declaration of the frame shows precisely which attributes can be changed within the entity, leaving the rest of the entity untouched.

If more than one attribute of the same entity type is defined, we need to impose an invariant stating that their domains are the same, in order to make sure that they all range over the same set of identifiers. So, for $att_i$ and $att_j$ forming the entity type $E$, we have:

\[
\text{[Id, Value}_i, \text{ Value}_j]
\]
\[
\text{att}_i : \text{Id $\leftrightarrow$ Value}_i
\]
\[
\text{att}_j : \text{Id $\leftrightarrow$ Value}_j
\]
\[
\text{con KnownE : } \forall \text{ Id}
\]
\[
\text{and dom (att}_i) = \text{ dom (att}_j) = \text{ KnownE}
\]

Entities associate through relationships and these can be modeled in terms of mathematical relations between the sets of identifiers. Thus, a relationship $R$ between two entities $E_i$ and $E_j$ with sets of identifiers $Id_i$ and $Id_j$, respectively, can be represented as:

\[
\text{var R : Id}_i$ $\leftrightarrow$ Id$;_j$
\]
\[
\text{and dom (R$\_\$) $\subseteq$ KnownE}_i \land \text{ ran (R$\_\$) $\subseteq$ KnownE}_j
\]

The invariants on the domain and range of the relation will be trivially maintained, since we do not change $R$ in our processes. If, in some particular situation, the whole set of identifiers need to be part of the relation, then we have $\text{ dom (R$\_\$) } = \text{ KnownE}_i$ (similarly, we may also have $\text{ ran (R$\_\$) } = \text{ KnownE}_j$).

As mentioned before, we concentrate on business processes that perform updates on databases, in the sense that the attributes of one (or more) entities can be changed, depending on certain conditions. We do not treat specifications that change the set of identifiers. Thus, our processes will always represent updates to the set of values of the attributes. An update to an attribute $att$ overrides the corresponding mapping with the set of maplets satisfying a given condition. This can be represented in the following way:

\[
att = att_0 \oplus \{id_1 : \text{KnownE}_1; \ldots, id_n : \text{KnownE}_n | \text{ cond$\circ$ id}_i \leftrightarrow \text{ exp}\}
\]

where we assume that $id_i$, for some $i \in 1 \ldots n$, ranges over the set of identifiers of $att$. $att_0$ represents the initial value of the attribute. $\text{ cond }$ and $\text{ exp }$ may involve attributes of entity types different from the one containing $att$.

Our refinement steps consist of eliminating set comprehensions of the above form by transforming them into singleton sets. Then, an update of the form $att = att_0 \oplus \{id \leftrightarrow exp\}$ can be translated into the assignment $att[id] = exp$. When more than one element of an entity type is to be updated, the refinement process will generate an iteration.
A conditional update involving a single maplet will be written as

\[ \text{att} = \text{att}_0 \oplus \{ \text{cond} \cdot \text{id} \mapsto \text{exp} \} \]

which is an abbreviation of

\[
\begin{align*}
\text{cond} \Rightarrow & \quad \text{att} = \text{att}_0 \oplus \{ \text{id} \mapsto \text{exp} \} \\
\neg \text{cond} \Rightarrow & \quad \text{att} = \text{att}_0
\end{align*}
\]

This will give rise to the program fragment:

\[
\begin{align*}
\text{if } & \quad \text{cond} \rightarrow \text{att}[\text{id}] := \text{exp} \\
& \quad \Box \neg \text{cond} \rightarrow \text{skip} \\
\text{fi}
\end{align*}
\]

To exemplify the idea of updates as function overridings, consider the following process:

Every customer should have sales reset to 0 and balance, if negative, reset to 0

where customer is an entity and balance and sales are two of its attributes.

Using the notations of the Refinement Calculus and the conventions introduced above we can formalise these requirements:

\[
\begin{align*}
\text{[CustomerId, Sales, Balance]} \\
\text{var customer_balance : CustomerId } & \rightarrow \text{Balance} \\
\text{var customer_sales : CustomerId } & \rightarrow \text{Sales} \\
\text{con KnownCustomer : } & \text{F CustomerId;} \\
\text{and } & \text{dom customer_balance } = \text{dom customer_sales } = \text{KnownCustomer } \\
\text{customer_balance, } & \text{customer_sales : } \\
\text{[true} & \text{, customer_balance } = \text{customer_balance}_0 \oplus \\
& \{ \text{cid : KnownCustomer } \mid \text{customer_balance}_0 \text{ cid } < 0 \cdot \text{cid } \mapsto 0 \} \lor \\
& \text{customer_sales } = \text{customer_sales}_0 \oplus \\
& \{ \text{cid : KnownCustomer } \cdot \text{cid } \mapsto 0 \}
\end{align*}
\]

We assume that the sets CustomerId, Sales and Balance are given. Because we do not change the value of identifiers, we declare KnownCustomer as constant; customer_balance and customer_sales are variables, and also part of the frame, which means that their values may change.

From the specification given above, we can apply the laws of the Refinement Calculus in order to generate code. The final code will be an extension to Dijkstra's guarded-command language, since we treat sets as variables and, consequently, the respective set operations (set difference \( \setminus \), set union \( \cup \), etc). Furthermore, we use a generalised assignment statement \( \text{el} : \in \text{set} \) to select an arbitrary element from a given set. The reason why we consider these abstract operations as code is that database query languages usually include statements which provide a similar level of abstraction; this is illustrated in the next section.

Due to the fact that all our updates need to have access to some entity type, it still remains to be described how this access can be done. The next section shows some rules for accessing entity types and their respective proofs.

### 3.2 Refinement Laws for Accessing the Database

In a PAD, we have two basic commands to access (or "read") the entity types: one for selecting a set of identifiers matching some condition and another for selecting one identifier (if there is any) for which the condition is true. The rest of this section is dedicated to presenting and proving laws for introducing
each of these two constructs\(^2\). It is important to highlight that these laws work as abbreviations for a set of refinement steps using the existing laws in the Refinement Calculus. The abbreviated versions lead to conciser and more elegant derivations.

3.2.1 Read Introduction

The problem of translating the effect of our first access statement into the Refinement Calculus can be divided into two subtasks: the first one checks whether or not, for some entity type \(E\), there exists at least one identifier whose value matches some given condition. For doing that, it is necessary to introduce two variables to our code: one will tell if such an identifier has been found (\(fnd\), holding value true or false) and the other will keep the value of the selected identifier (\(sel\)), so that its value can be incorporated into our specification; the second subtask introduces an alternation based on the value of \(fnd\). Our law for introducing the effect of read statement is as follows:

**Read Introduction** (ReadI)

\[
\begin{align*}
\text{w : [pre, post]} &
\leq \\
&[[\text{var } sel : KnownE; fnd : BOOL} \\
&\quad \quad \text{sel}, \quad ((\exists \text{id : KnownE } | \text{cond(id) } \land \text{sel } = \text{id}) \land \text{fnd}) \lor \\
&\quad \quad \text{fnd} : \quad \text{pre}, \quad ((\forall \text{id : KnownE } | \neg \text{cond(id)}) \land \neg \text{fnd}) \\
&\quad \quad \text{if } fnd \rightarrow w : [[(\exists \text{id : KnownE } | \text{cond(id) } \land \text{pre, post}] \\
&\quad \quad \quad \text{else } fnd \rightarrow w : [[(\forall \text{id : KnownE } | \neg \text{cond(id)}) \land \text{pre, post}] \\
&\quad \quad \quad \text{fi.}]]
\end{align*}
\]

This law can be justified in terms of more basic laws of the Refinement Calculus, as shown in [2]. Its validity is independent of \(w\), \(pre\) and \(post\), since its application changes only \(sel\) and \(fnd\), which are local variables. In practice, however, it will be useful only when \(post\) mentions the entity \(E\).

As the specification with label \((i)\) is concerned with searching for an identifier \((id)\) whose value matches some given condition—if there is such an identifier—we can derive the corresponding code for this general case, rather than for each instance that will appear in our examples (as a consequence of applying the law ReadI). This is formally done in [2]. Here, we are concerned with the final code, which is given below. The idea is to copy the set of identifiers \(KnownE\) into a temporary variable, and search for an element matching the condition; each element tested is taken from the set; the loop for searching in the set ends either when the set is empty or when one element satisfying the condition has been found.

\[
\begin{align*}
[[\text{var } sel : KnownE; fnd : BOOL} \\
[[\text{var } sid : F KnownE} \\
\quad \quad \text{sid, fnd } := \text{KnownE, false;} \\
\quad \quad \text{do (ssid } \neq \emptyset \land \neg fnd) } \rightarrow \\
\quad \quad \quad [[\text{var } el : KnownE} \\
\quad \quad \quad \quad \quad \quad \text{el } \in \text{sid;} \\
\quad \quad \quad \quad \quad \quad \text{if } \neg \text{cond(el) } \rightarrow \text{sel, fnd } := \text{el, true} \\
\quad \quad \quad \quad \quad \quad \text{fi;} \\
\quad \quad \quad \quad \quad \quad \text{if (cond(el)) } \rightarrow \text{skip} \\
\quad \quad \quad \quad \quad \text{fi;} \\
\quad \quad \quad \quad \quad \text{sid } := \text{sid } \setminus \{\text{el}\}]]
\end{align*}
\]

\(^2\)Actually, we do not introduce new statements to perform these operations. Rather, we show how their effect can be described in terms of the predicate calculus and the guarded command language.
3.2.2 Read Each Introduction

The second statement for accessing the database (Read Each) filters the entity type, generating a set containing the elements that match the given condition; this set is then used as the basis for the update.

As in the previous law, we split our problem into two: first, we generate a set that contains the elements filtered by the condition (if there are any); the second subproblem consists of some actions that are performed taking the newly generated set into consideration. Thus, the rule for introducing the Read Each is as follows:

**Read Each Introduction (ReadEachI)**

\[ w : [pre, post] \]
\[ \subseteq \]
\[ \{ [var sid : F KnownE \bullet ] \]
\[ \{ sid, tsid := \emptyset, KnownE; \}
\[ \{ do tsid \neq \emptyset \rightarrow \}
\[ \{ [var el : KnownE \bullet ] \]
\[ \{ el \in tsid; \}
\[ \{ if \ cond(el) \rightarrow sid := sid \cup \{ el \} \}
\[ \{ \neg (cond(el)) \rightarrow skip \}
\[ \{ tsid := sid \setminus \{ el \} \} \}
\[ od] \]

The above law follows directly from “varI” and “semI”.

As in the previous case, the specification with label (i) can be refined into the statements that generate the subset of KnownE containing the elements matching the condition. The formal derivation is presented in [2]; below, we give the final result:

\[ \{ [var sid : F KnownE \bullet ] \]
\[ \{ [var tsid : F KnownE \bullet ] \]
\[ \{ sid, tsid := \emptyset, KnownE; \}
\[ \{ do tsid \neq \emptyset \rightarrow \}
\[ \{ [var el : KnownE \bullet ] \]
\[ \{ el \in tsid; \}
\[ \{ if \ cond(el) \rightarrow sid := sid \cup \{ el \} \}
\[ \{ \neg (cond(el)) \rightarrow skip \}
\[ \{ sid := sid \setminus \{ el \} \} \}
\[ od] \]

4 Example

This section presents an example showing code derivation from the specification of a business process. The example chosen is intentionally small, covering simple updates to the database. This allows us to show a complete derivation for the problem, where each step is justified by the application of one or more laws introduced in the previous section. Additional examples illustrating derivations for other kinds of database constructs can be found in [2].

For conciseness, the names of the variables and types that will take part on the derivations have been abbreviated by their initials (for example, `soed` for `sales_order_entered_date` and `KSO` for `KnownSalesOrder`).

Consider the following informal specification:

For all orders that have been requested before the current date, `entered_date` is set to the current date, and, if those orders have been placed by an active customer (ie, with `status` = “A”), they also have the `requested_date` set to the current date.

We assume that the following sets are given:

\[ [SalesOrderId, CustomerId, Date, Status] \]

Because `sales_order_entered_date` and `sales_order_requested_date` are part of the same entity type—`SalesOrder`—their domains have to be the same. So, we have:

1251
We will use \( cd \) to represent the current date.

The process specification can now be formally specified as:

\[
\begin{align*}
\text{soed, sord} & : SalesOrderId \rightarrow \text{Date}; \\
\text{con} \text{KSO} & : \forall \text{SalesOrderId}; \\
\text{and dom soed} & = \text{dom sord} = \text{KSO} \\
\text{var cs} & : \text{CustomerId} \rightarrow \text{Status}; \\
\text{con} \text{KC} & : \forall \text{CustomerId}; \\
\text{and dom cs} & = \text{KC}
\end{align*}
\]

The set of SalesOrder identifiers matching the condition is generated by applying the law "ReadEachI":

\[
\begin{align*}
\subseteq \quad \text{"ReadEachI"} \\
& \quad \quad \quad \mid [\text{var ssid} : \forall \text{KSO} \cdot \\
& \quad \quad \quad \quad \text{ssid} : [\text{true}, \text{ssid} = \{ \text{sid} : \text{KSO} | \text{sord} \text{ sid} < \text{cd} \}] ; \\
& \quad \quad \quad \quad \quad \text{ssid} = \{ \text{sid} : \text{KSO} | \text{sord} \text{ sid} < \text{cd} \} \\
& \quad \quad \quad \quad \text{soed, sord} : [\text{true}, \text{soed} = \{ \text{sid} : \text{KSO} | \text{sord} \text{ sid} < \text{cd} \cdot \text{sid} \rightarrow \text{cd} \} \wedge \\
& \quad \quad \quad \quad \quad \text{sord} = \{ \text{sid} : \text{KSO} | \text{sid} \rightarrow \text{cd} \}] \quad (i) \\
& \quad \quad \quad \quad \text{sid IsPlBy cid \wedge cs cid = \text{"A"} \cdot \text{sid} \rightarrow \text{cd}} \\
& \quad \quad \quad \quad \text{soed} = \text{soed} \bot = \{ \text{sid} : \text{KSO} | \text{sord} \text{ sid} < \text{cd} \cdot \text{sid} \rightarrow \text{cd} \} \wedge \\
& \quad \quad \quad \quad \quad \text{sord} = \{ \text{sid} : \text{KSO} | \text{sid} \rightarrow \text{cd} \}] \\
& \quad \quad \quad \quad \text{soed} = \text{soed} \bot = \{ \text{sid} : \text{KSO} | \text{sord} \text{ sid} < \text{cd} \cdot \text{sid} \rightarrow \text{cd} \} \\
& \quad \quad \quad \quad \quad \text{sord} = \{ \text{sid} : \text{KSO} | \text{sid} \rightarrow \text{cd} \}] \\
& \quad \quad \quad \quad \text{soed} = \text{soed} \bot = \{ \text{sid} : \text{KSO} | \text{sord} \text{ sid} < \text{cd} \cdot \text{sid} \rightarrow \text{cd} \} \\
& \quad \quad \quad \quad \quad \text{sord} = \{ \text{sid} : \text{KSO} | \text{sid} \rightarrow \text{cd} \}] \\
\end{align*}
\]

The code for the first specification above is given in the end of this example.

The pre-condition of (i) implies the next pre-condition with the introduction of the fresh constants \( \text{soedx} \) and \( \text{sordx} \):

\[
\subseteq \quad \text{"conI"} \\
\mid [\text{con soedx, sordx} : \text{SalesOrderId} \rightarrow \text{Date} \cdot \\
& \quad \quad \quad \mid (\forall \text{sid} : \text{KSO} \cdot \text{sid} \in \text{ssid} \Rightarrow \text{sordx} \text{ sid} < \text{cd}) \wedge \text{ssid} \subseteq \text{KSO} \wedge \\
& \quad \quad \quad \quad \text{soed} = \text{soed} \bot = \{ \text{sid} : \text{KSO} \setminus \text{ssid} | \text{sordx} \text{ sid} < \text{cd} \cdot \text{sid} \rightarrow \text{cd} \} \wedge \\
& \quad \quad \quad \quad \quad \text{sord} = \{ \text{sid} : \text{KSO} \setminus \text{ssid} | \text{cid} : \text{KC} | \text{sordx} \text{ sid} < \text{cd} \wedge \\
& \quad \quad \quad \quad \quad \quad \text{sid IsPlBy cid \wedge cs cid = \text{"A"} \cdot \text{sid} \rightarrow \text{cd}} \} \wedge \\
& \quad \quad \quad \quad \quad \quad \text{soed} = \text{soed} \bot = \{ \text{sid} : \text{KSO} \setminus \text{ssid} | \text{sid} \rightarrow \text{cd} \} \\
& \quad \quad \quad \quad \quad \quad \text{sord} = \{ \text{sid} : \text{KSO} | \text{sid} \rightarrow \text{cd} \}] \\
& \quad \quad \quad \quad \text{soed} = \text{soed} \bot = \{ \text{sid} : \text{KSO} | \text{sordx} \text{ sid} < \text{cd} \cdot \text{sid} \rightarrow \text{cd} \} \\
& \quad \quad \quad \quad \quad \quad \text{sord} = \{ \text{sid} : \text{KSO} | \text{sid} \rightarrow \text{cd} \}] \\
& \quad \quad \quad \quad \text{soed} = \text{soed} \bot = \{ \text{sid} : \text{KSO} | \text{sordx} \text{ sid} < \text{cd} \cdot \text{sid} \rightarrow \text{cd} \} \\
& \quad \quad \quad \quad \quad \quad \text{sord} = \{ \text{sid} : \text{KSO} | \text{sid} \rightarrow \text{cd} \}] \\
\end{align*}
\]

Note that it was possible to use \( \text{soedx} \) and \( \text{sordx} \) also in the post-condition, as \( \text{soed} = \text{soedx} \) and \( \text{sord} = \text{sordx} \) in the pre-condition.

It is now necessary to go through the selected identifiers, and execute the body of the read statement; in order to do that, an invariant needs to be stated and the post-condition should express the state of the guard of the loop, after its execution:

\[
\subseteq \quad \text{"sp"}
\]
As long as \( \text{ssid} \) is not empty, the loop should continue; at each iteration, \( \#\text{ssid} \) decreases by one, as one element is extracted from the set:

\[
\begin{align*}
\text{do} & \quad \text{ssid} \neq \emptyset \\
& \quad \text{ssid, soed, sord :} \ [\text{ssid} \neq \emptyset \land I, \ 1 \land 0 \leq \#\text{ssid} < \#\text{ssido}] \\
& \quad \text{od}
\end{align*}
\]

\( \text{el} \) is subtracted from \( \text{ssid} \)—after the update has taken place—within the loop, making \( \#\text{ ssid} \) to decrease:

\[
\begin{align*}
\text{fassI} & \quad \text{ssid} \neq \emptyset \land I \\
& \quad \text{ssid, el, soed, sord :} \ [\text{ssid} \neq \emptyset \land I, \ 1 \land 0 \leq \#\text{ssid} < \#\text{ssido} \land \text{el} \in \text{ssido} \land \text{ssid} = \text{ssido}\setminus\{\text{el}\}] \\
& \quad \text{ssido} := \text{ssido}\setminus\{\text{el}\}
\end{align*}
\]

\[
\begin{align*}
\text{contract frame} & \quad + \quad \text{“sp”} \\
& \quad \text{el, soed, sord :} \ [\text{el} \in \text{ssido} \land \text{ ssid} \neq \emptyset \land I, \ 1 \land 0 \leq \#\text{ssido} < \#\text{ssido} \land \text{soed} = \text{soed}\setminus\{\text{el}\} \land \text{sord} = \text{sord}\setminus\{\text{el}\} \land \text{cid} = \text{KC} \land \text{ ssid} \neq \emptyset \land I \land 0 \leq \#\text{ssido} < \#\text{ssido} \land \text{ssido} := \text{ssido}\setminus\{\text{el}\} \\
& \quad \text{el} \in \text{ssido} \land 0 \leq \#(\text{ssido}\setminus\{\text{el}\}) < \#\text{ssido} \\
& \quad \text{ssid} := \text{ssid}\setminus\{\text{el}\}
\end{align*}
\]

For each identifier \( \text{el} \) belonging to \( \text{ssid} \), the update is done:

\[
\begin{align*}
\text{leading (generalised) assignment} & \quad + \quad \text{contract frame}
\end{align*}
\]
el :∈ ssid;
soed, = soedX + {el ↔ cd}∧
sord : [true, soed = soedX + {cid : KC | el IsPlBy cid ∧ cs cid = “A” • el ↔ cd}]

“leading assignment” + “contract frame”
soed[el] := cd;
sord : [true, sord = sordX + {cid : KC | el IsPlBy cid ∧ cs cid = “A” • el ↔ cd}]

Customer needs to be accessed, in order to have the other conditions checked:

“ReadI”
[[var sel : KC; fnd : BOOL •
  sel, = [(∃ cid : KC | el IsPlBy cid ∧ cs cid = “A” • sel = cid ∧ fnd)∨
  fnd : [true, ((∀ cid : KC • ¬(el IsPlBy cid ∧ cs cid = “A”) ∧ ¬fnd)) ;
  if fnd →
    sord : [(∃ cid : KC | el IsPlBy cid ∧ cs cid = “A” • sel = cid ∧ fnd)]
  ⊢ ¬fnd →
    sord : [(∀ cid : KC • ¬(el IsPlBy cid ∧ cs cid = “A”) ∧ ¬fnd)]
  fi]]

(ii) “sp”
(sord : [(∃ cid : KC | el IsPlBy cid ∧ cs cid = “A” • sel = cid)]

(iii) “assI”
(sord[el] := cd)

(iii) skip

The collected code for this derivation is given below.

[[var ssid : F KSO •
  [[var tssid : F KSO •
    tssid, ssid := KSO, Ø;
    do tssid ≠ Ø →
      [[var elc : KSO •
        elc ∈ tssid;
        if sord elc < cd → ssid := ssid ∪ {elc}
        ⊢ sord elc ≥ cd → skip
        fi;
        tssid := tssid \{elc}]]]
  od;
]]
  do ssid ≠ Ø →
    [[var el : KSO •
      el :∈ ssid;
      soed[el] := cd;
      [[var sel : KC, fnd : BOOL •
        var scid : F KC •
        scid, fnd := KD, false;]]]}}
do (scid ≠ ∅) →
| (a) ele scid; ele
| if el IsPlBy ele → sel, fnd := ele, true
| □ ¬(el IsPlBy ele) → skip
| fi;
| scid := scid\{ele\}
| od;
| if fnd → sord[el] := cd
| □ ¬fnd → skip
| fi;
| ssid := ssid\{el\}
| od

5 Conclusion

This work is an attempt of using Morgan’s Refinement Calculus to derive implementations from given specifications of business processes; more specifically, processes defined using the IE method, working on a database environment.

We have shown how to provide a stepwise derivation of programs. Our first step was the selection of a subset of the refinement laws (described in Section 2) relevant to our application. Then we have defined a simple mathematical model in which we can formalise requirements describing business processes. This was achieved by defining entity attributes as partial functions and updates as function overridings. Furthermore, some laws were suggested in connection with access to and update of entity types, depending on a given condition. These new laws are derivable [2] from more basic laws of the Refinement Calculus. With this toolkit in hands, we could derive programs which, by construction, satisfy the given specifications. The problem treated in Section 3 showed how to generate code for updates on multiple attributes.

As the need for designing large systems—very often in a database environment—is increasing every day, it is extremely important to be able to derive programs in a stepwise manner. The idea of using the Refinement Calculus showed to be an appropriate one, as each transformation is justified by a refinement law, which guarantees that the derived program will be correct. This stepwise approach contrasts with the automatic derivation of the PAD, where the analyst/designer can’t interfere on the way the final code is generated. Furthermore, doing program refinements it is possible to have more than one program for the same specification and the fundamental issue of efficiency could be addressed.

Still, much work can be done: new constructs can be translated into the mathematical notation, new refinement laws can be suggested and even refinements can be mechanised. These are topics for future work.

References