Formal Development of Databases in ASSO and B

Brian Matthews[1] and Elvira Locuratolo[2]

[1] CLRC, Rutherford Appleton Laboratory Didcot, OXON, OX11 OQX, U.K.
Telephone: +44 1235 446648. Fax: +44 1235 445831
bmm@inf.rl.ac.uk
[2] Istituto di Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche,
Via S. Maria, 46 - 56126 Pisa, Italy
Telephone: +39 50 593403. Fax: +39 50 554342
locuratolo@iei.pi.cnr.it

Abstract. ASSO is a formal methodology for conceptual databases design which uses aspects of the B method. In this paper, we discuss the formal relationship between specification in ASSO and in B, defined in terms of a translation which we prove to be sound. Further we go onto discuss refinement in ASSO, which separates behavioural and data refinement, their formal relationship and their relation to refinement in B. In this manner we can use B theory and tools to support database development in ASSO.

1 Introduction

Formal methods such as B, Z and VDM have been developed to be applicable to a broad range of computing problems. However, there are advantages to be gained from methods specialised to particular application domains. The restricted nature of the systems to be developed may impose conditions on the methods, which can then be tailored to take advantage of the properties of that domain. Such specialised methods would also be more acceptable to developers accustomed to working within the culture of the application.

ASSO [7] is a formal design methodology for conceptual database design which has been designed to combine ease of modification with efficiency of implementation whilst ensuring specification consistency and design correctness. ASSO has been developed by integrating features of two methods. Firstly the Partitioning Method [15], a formal relation between *semantic data models* [11] a high-level view where classes are linked through specialisations within a single hierarchy, and *object models* [1, 5], where classes are disjoint. ASSO extends this with a notion of behavioural specialisation which allows both structural and behavioural aspects of database applications to be specified using a *Structured Database Schema* [3, 14]. Secondly, the B Method [2], one of the most highly developed

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 388–410, 1999.

formal methods for general software engineering currently being promoted for commercial use. Each ASSO schema can be translated into an Abstract Machine and the formal semantics of the ASSO schemas is given exploiting the pre-existing abstract machine semantics [3]. In this paper we assume that the reader is familiar with the notions of B.

The direct use of B for developing database applications has disadvantages. The B method lacks the abstraction mechanisms supported by database conceptual languages, and its refinement has not been designed to obtain efficient database implementations. Within ASSO, both specifications and refinements are presented using a formal conceptual design notation which uses terms familiar to database developers. The formal semantics of the method exploit the nature of transactions, which preserve the integrity of the database, to simplify some proof obligations. The Partitioning Method decomposes the schema towards a logical schema without further proof obligations.

The Structured Database Schema can be translated systematically into *Abstract Machines*, and the consistency proofs of the ASSO conceptual schema can be split into small B proofs [17, 20]. Further, the schema transformations of ASSO refinement are also particular steps of B-refinement [18]. In this paper, we formalise this translation, via a reformulated semantics for ASSO. This is equivalent to that given in [3] but it is given in terms of the language of the semantics of B [2]. This semantics is then extended to cater for the structuring mechanisms for ASSO, especially *is-a**. As the semantics of ASSO are now given in the same framework as B, it is then straightforward to demonstrate that the translation is sound. Further, we discuss refinement in ASSO, which combines behavioural refinement with the Partitioning Algorithm. Using the same semantic framework we then give some results on refinement, and discuss its relationship with refinement in B.

This paper is organised as follows. In section 2 we give a description of ASSO and give its semantics in terms of weakest preconditions. In section 3 we describe the translation of ASSO to B and prove the soundness of this relation. Refinement in ASSO and its relation to refinement in B is discussed in section 4.

2 Specification in ASSO

The *Structured Database Schema* (SDBS) is the model which supports ASSO specifications. It exploits the concepts of *class*, and *is-a* relationships* between classes. A *class C* is an entity defining both the static and behavioural aspects of a set of entries in the database. The static aspects are a set of *objects* and a set of *attributes* associated with those objects, described using sets and functions. The dynamic aspects comprise a set of state transformations, the *operations*, with a distinguished operation *initialisation* defining the initial state of the class. Operations are defined by applying constructors to basic operations. In this paper, we use a generic class C, on given set S, with attributes: $a_1 : t_1 \ldots a_n : t_n$

initialisation C, a_1, ..., $a_n := I_C$, I_1 ... I_n, application constraint $P(C, a_1, \ldots, a_n)$,and operations op_1C, ..., op_kC. Note that in ASSO the notation $a : t$ means an attribute a has a type t in a similar sense to an object member variable. It should not be confused with the ASCII B notation's use of the colon for set membership $\in$. For a class C, the following basic operations are available:

ADD C $(x, v_1, \ldots, v_n)$	Insert object x with attribute values $v_1, \ldots, v_n$ into C.
REM C (x)	Remove object x from C.
SKIP C	Does nothing in C.
CHANGE C a_i (x,v_i)	Update value of attribute a_i of object x in C to v_i.

The weakest precondition semantics for a base operation *op* in C are given via the equivalence on any arbitrary predicate R:

$$[op\ C\ (par\_list)\ ]R \Leftrightarrow \exists\, v' \cdot (R' \wedge \mathrm{PRE}_C(op\ C\ (par\_list)) \wedge prd_C\ (op\ C\ (par\_list)))$$

where v' is a list of primed variables of the class, $C', a'_1, \ldots a'_n$ and R' is the result of priming all variables in R, representing the variables of C after the operation. We define the relationship between the pre and post variables using the before-after predicate prd_C a predicate which holds for pairs of states similar to that used to specify operations in VDM and Z. In B, the predicate is subscripted with a variable x; in ASSO we subscript the predicate with a class C. We then define the predicate $prd_C(S)$ for an operation S following ([2], §6.3.3) as:

$$prd_C(S) \Leftrightarrow \neg[S](C' \neq C) \vee (a'_1 \neq a_1) \vee \ldots \vee (a'_n \neq a_n)$$

Thus we have for the base operations of ASSO;

$$\begin{aligned}
&prd_C(\text{ADD } C\ (x,v_1,\ldots,v_n)) = \\
&\qquad (C' = C \cup \{x\}) \wedge (a'_1 = a_1 \lhd \{(x,v_1)\}) \wedge \ldots \wedge (a'_n = a_n \lhd \{(x,v_n)\}) \\
&prd_C(\text{REM } C\ (x)) = (C' = C - \{x\}) \wedge (a'_1 = \{x\} \lhd a_1) \wedge \ldots \wedge (a'_n = \{x\} \lhd a_n) \\
&prd_C(\text{SKIP } C) = (C' = C) \wedge (a'_1 = a_1) \wedge \ldots \wedge (a'_n = a_n) \\
&prd_C(\text{CHANGE } C\ a_i\ (x,v_i)) = \\
&\qquad (C' = C) \wedge (a'_1 = a_1) \wedge \ldots \wedge (a'_i = a_i \lhd \{(x,\, v_i)\}) \wedge \ldots \wedge (a'_n = a_n)
\end{aligned}$$

We differ from [3] by explicitly including the *precondition* of the base operations:

$$\begin{aligned}
&PRE_C(\text{ADD } C\ (x,v_1,\ldots,v_n)) = x \in S \wedge x \notin C \wedge v_1 \in t_1 \wedge \ldots \wedge v_n \in t_n \\
&PRE_C(\text{REM } C\ (x)) = x \in C \\
&PRE_C(\text{SKIP } C) = true \\
&PRE_C(\text{CHANGE } C\ a_i\ (x,v_i)) = x \in C \wedge v_i \in t_i
\end{aligned}$$

The following constructors can be applied recursively to the basic operations:

PRE P **THEN** *op* C (*par_list*) **END**	Pre-conditioning.
$P \rightarrow$ *op* C (*par_list*)	Guarding.
CHOICE op_1 C (*par_list*) **OR** op_2 C (*par_list*) **END**	Choice.
ANY y **WHERE** P **THEN** *op* C (*par_list*) **END**	Unbounded-choice.

Here, P is a predicate on the variables of class C, *op* C (*par_list*) op_1 C (*par_list*) and op_2 C (*par_list*) are operations on C and y is a variable distinct from the variables of C. Pre-conditioning specifies a the *preconditioned transformation*: *op* C (*par_list*) for the states satisfying P and is undetermined otherwise. Guarding specifies a *partial transformation*: *op* C (*par_list*) for the states satisfying P and abort otherwise. Choice specifies a *non-deterministic transformation* between op_1 C (*par_list*) and op_2 C (*par_list*). Unbounded-choice specifies an *unbounded non deterministic transformation* defined by replacing any value of y satisfying P in *op* C (*par_list*).

The weakest precondition semantics for a constructed operation *op* in C are:

$$[op\ C\ (par\_list)\ ]R \Leftrightarrow \exists v' \cdot (R' \wedge prd_C\ (op\ C\ (par\_list)))$$

The before-after predicate for the operation constructors is as follows:

$$prd_C\ (\textbf{PRE}\ P\ \textbf{THEN}\ op\ C\ (par\_list)\ \textbf{END}) = P \Rightarrow prd_C\ (op\ C\ (par\_list))$$
$$prd_C\ (P \rightarrow op\ C\ (par\_list)) = P \wedge prd_C\ (op\ C\ (par\_list))$$
$$prd_C\ (\textbf{CHOICE}\ op_1\ C\ (par\_list)\ \textbf{OR}\ op_2\ C\ (par\_list)\ \textbf{END}) =$$
$$prd_C\ (op_1\ C\ (par\_list)) \wedge prd_C\ (op_2\ C\ (par\_list))$$
$$prd_C\ (\textbf{ANY}\ y\ \textbf{WHERE}\ P\ \textbf{THEN}\ op\ C\ (par\_list)\ \textbf{END}) =$$
$$\exists y \cdot (P \wedge prd_C\ (op\ C\ (par\_list)) \text{ if } y \text{ is not in } C'.$$

A class has a set of *class constraints*:

$$class\text{-}constraints_C = C \subseteq U \wedge a_1 \in C \rightarrow t_1 \wedge \ldots \wedge a_n \in C \rightarrow t_n$$

That is that all objects in C must be of the given set and every attribute on C can be regarded as a *total function* on the class. The class constraints are preserved by the valid operations on the class.

Additional (explicit) application constraints may be associated with the class in the form of a predicate I_C which have to be preserved by all class's operations. Given operation *op* with precondition P the consistency obligation is given by:

$$I_C \wedge P \Rightarrow [S]I_C$$

Note that this is the same as the proof obligation on invariants in B. However, the class constraints do not need to be proved.

Further, in class C there is a distinguished operation $Init_C$ which initialises the class and can assign *any* value to the class variables. Consequently, the class constraints cannot be guaranteed to hold for $Init_C$ and thus there is an initialisation obligation to show that they are established.

$$[Init_C\ ](I_C \wedge \textit{class-constraints}_C)$$

Within a SDBS, a class can be either the root of a specialisation hierarchy or defined through the *is-a** relationship with a superclass. The *is-a** relationship extends the standard *is-a* relationship by allowing the inheritence of both attributes and operations from a superclass [3].

Given the specialised class C_2 *is-a** C_1, then the following properties hold:

- Class C_2 *is-a** C_1 inherits both attributes and operations from class C_1.
- Inherited operations are specialised on C_2, that is restricted to the subclass. The initialisation and the operations on C_2 which insert objects are explicitly specialised, whereas other operations are implicitly specialised.
- Objects of C_2 are a subset of C_1 objects.
- Class C_2 can have specific attributes and/or operations.

An *inherited* operation from C_1 on C_2 is the parallel composition [8] of the operation on C_1 with the same operation *specialised* on C_2. The *is-a** relation imposes the following constraint:

$$\begin{aligned}\textit{is-a-constraint}_{C_2 \textit{is-a}^* C_1} = {} & C_2 \subseteq C_1 \wedge C_2 \lhd a_1 \in C_2 \rightarrow t_1 \wedge \ldots \wedge \\ & C_2 \lhd a_n \in C_2 \rightarrow t_n\end{aligned}$$

where $a_1{:}t_1 \ldots a_n{:}t_n$ are the attributes of C_1.

The following syntactic forms are used to specify ASSO schemas:

class *name* **of** *given-set* **with** (*attr-list; const ; oper-list*)
class *name* **is-a*** *name* **with** (*attr-list; const ; oper-list*)

The former is the *base* constructor of the ASSO schema classes used to specify classes at the top of the specialisation hierarchy. The latter is used to specify classes in *is-a** relationship with other classes. Within these forms, *name* denotes the class name, *attr-list* the attributes of the class, *constr* application constraints on the class, and *oper-list* the operations including the initialisation. The class constraints are implicitly specified with the class constructor.

Definition 1. *A Structured Database Schema is a list of classes such that the top class is defined through a base constructor and the remaining classes through* is-a* *constructors.*

If C_1 *is-a** C_2 then the weakest precondition for inherited operations of C_1 *is-a** C_2 is given by:

$$\begin{aligned}[\ \textit{op}\ C_1\ \textit{is-a}^*\ C_2\ (\textit{par\_list})\ ]R \Leftrightarrow {} & \exists\, v'_R \cdot (\textit{prd}_C\ (\textit{op}\ C_1\ (\textit{par\_list})) \wedge \\ & \textit{prd}_C\ (\textit{op}\ C_2\ (\textit{par\_list})) \wedge (\textit{is-a-constraints} \Rightarrow \textit{is-a-constraints}') \wedge R')\end{aligned}$$

If application constraints involves only variables in single classes, then an SDBS is a set of classes linked through the *is-a** relation, and is consistent (that is satisfies all consistency obligations) if and only if each class is consistent.

2.1 A Set Theoretic Model of ASSO

To simplify proofs, we also give a set theoretic model of the semantics of ASSO, following that for B as given in [2] §6.4.1.

The termination predicate $trm(S)$ on operations is defined ([2], §6.3.1) as:

$$trm(S) \Leftrightarrow [S](x = x)$$

That is if after the operation *true* can be established, then the operation terminates. From this derives the definitions of the termination predicate for the base operations of ASSO. They all terminate within their precondition:

$$trm(\text{ ADD } C\ (x, v_1, \ldots, v_n)) = (x \notin C)$$
$$trm(\text{ REM } C\ (x)) = (x \in C)$$
$$trm(\text{ SKIP } C\ ) = true$$
$$trm(\text{ CHANGE } C\ a_i\ (x, v_i)) = (x \in C)$$

We then define the termination predicate recursively for the constructors.

$$trm\ (\textbf{PRE}\ P\ \textbf{THEN}\ op\ C\ (par\_list)\ \textbf{END}) = P \wedge trm\ (op\ C\ (par\_list))$$
$$trm\ (P \rightarrow op\ C\ (par\_list)) = P \Rightarrow trm\ (op\ C\ (par\_list))$$
$$trm\ (\textbf{CHOICE}\ op_1\ C\ (par\_list)\ \textbf{OR}\ op_2\ C\ (par\_list)\ \textbf{END}) =$$
$$trm\ (op_1\ C\ (par\_list)) \wedge trm\ (op_2\ C\ (par\_list))$$
$$trm\ (\textbf{ANY}\ y\ \textbf{WHERE}\ P\ \textbf{THEN}\ op\ C\ (par\_list)\ \textbf{END}) =$$
$$\forall y \cdot (P \Rightarrow trm\ (op\ C\ (par\_list))\ \textit{if } y \textit{ is not in } C'$$

Abrial imposes a condition that the types of *variables* do not change in an operation. This also holds in ASSO where variables are sets of class instances and total functions on those sets for class attributes. The condition becomes:

$$class\text{-}constraints_C \wedge trm\ (\ S\ ) \Rightarrow [\ S\ ]class\text{-}constraints_C$$

For a set theoretic characterisation of ASSO we follow B in giving two sets for each substitution: a *pre-condition set* $pre(S)$, representing the states where the preconditions of an operation holds; and a binary relation $rel(S)$ capturing the dynamics of the operation through the before-after predicate.

$$pre\ (\ S\ ) = \{C,\ a_1,\ \ldots,\ a_n |\ class\text{-}constraints_C \wedge trm\ (\ C\ )\ \}$$

$$rel\ (\ S\ ) = \{\ (C, C'),\ (a_1, a_1'),\ \ldots,\ (a_n, a_n')\ |\ class\text{-}constraint_C \wedge class\text{-}constraints'_C \wedge prd_C\ (S)\}$$

structured database schema
database
class *person* **of** *PERSON* **with** (*income*:*N*
constraints $\forall$ ***p*** **(*p*$\in$*person* $\Rightarrow$ *income*(*p*)$\geq$ 1000)**
init.person() = *person*, *income* := {},{}
new.person(*pers*, *i*) $\widehat{=}$
PRE *pers* $\in$ *PERSON* $-$ *person* $\wedge$ *i* $\geq$ 1000
THEN *ADD person*(*pers*, *i*)
END ;
del.person(*person*) $\widehat{=}$
PRE *pers* $\in$ *PERSON*
THEN *REM person*(*pers*)
END)
class *employee* **is-a*** *person* **with** (*salary*:*N*
constraints $\forall$ ***e*** **(*e*$\in$*employee* $\Rightarrow$ *salary*(*e*)$\geq$500)**
init.employee() $\widehat{=}$ *employee*, *salary* := {},{}
new.employee(*emp*,*s*) $\widehat{=}$
PRE *s* $\geq$ 500
THEN *ADD employee*(*emp*, *s*)
END)
class *student* **is-a*** *person* **with** (*matriculation*:*N*
constraints $\forall$ ***s1, s2*** **(*s1*$\in$*student* $\wedge$ *s2*$\in$*student* $\wedge$ *s*1 $\neq$ *s*2 $\Rightarrow$
matriculation(*s1*)$\neq$*matriculation*(*s2*))
init.student() $\widehat{=}$ *student*, *matriculation* := {},{}
new.student(*pers*) $\widehat{=}$
PRE *pers* $\in$ *PERSON* $-$ *student*
THEN
ANY *m* **WHERE** $m \in N \wedge m \notin \mathsf{ran}(matriculation)$
THEN *ADD student*(*pers*, *m*)
END
END)

Figure 1: An ASSO Specification

2.2 An Example ASSO Specification

Consider a database application with the following requirements [16].

– The database maintains a set of *persons* and their *income*, a subset of *employees* and their *salary*, and a subset of *students* and their *matriculation*.

- The *income* of a *person* is greater or equal to 1000; the *salary* of an *employee* is greater or equal to 500; a student has a unique matriculation number.
- Instances are added when a *new* person is inserted into the database and extended when a person is employed or when a person becomes a student.
- Information is removed from the database when an *employee* leaves the company and/ or when a *student* leaves or completes his or her studies.

From these requirements, we can derive an SDBS specification given in Figure 1. *del.employee* and *del.student* are operations inherited from the superclass without being explicitly specified, so *del.employee* on class *employee* results from the parallel composition of *del.person* with *del.person* implicitly specialised on the *employee*. *init.employee* and *init.student*, (respectively *new.employee* and *new.student*) are specialised; *new.employee* on class *employee* results from the parallel composition of *new.person* with *new.employee* specified on the *employee*.

As the constraints of the *is-a* relationship are preserved and the application constraints only involve separate classes [3], the consistency proof of the specified conceptual schema can be decomposed to those of the classes.

3 Relating the B-Method and ASSO

The relationship between ASSO and B is established via a formal translation from ASSO SDBSs into B machines, summarised as follows:

- Each class of a structured database schema can be translated into a B machine which preserve the class constraints and the *is-a** constraints.
- The consistency proof of a SDBS can be split into the proof of a set of small B consistency lemmas. Some lemmas are known to hold via the construction of the SBDS.

Given a class C we denote its translation into B by $T(C)$. We use similar notations for the translations of other ASSO constructs into B. The translation is shown to be correct by demonstrating that the semantics of an ASSO schema are the same as the semantics of the set of machines into which it is translated.

Definition 2. *Translation T is* sound *if for each ASSO class C on given set S:*

1. *For the class and attribute variables $C, a_1{:}t_1 \ldots a_n{:}t_n$ of C, there are translations such that: $T(C) \subseteq S$, $T(a_1) \in C \rightarrow T(t_1) \ldots T(a_n) \in C \rightarrow T(t_n)$.*
2. *For the explicit constraint P on the variables of C, there is a constraint $T(P)$ on the variables of $T(C)$ such that:*

$$\forall c \subseteq S, v_1 \in t_1 \ldots v_n \in t_n \cdot P(c, v_1, \ldots, v_n) \Leftrightarrow T(P(T(c), T(v_1), \ldots, T(v_n)))$$

3. *For each operation* op *in C, there is an operation T(*op*) in T(C) such that for any predicate R:* $[\ \text{op}\ (\text{par\_list})\ ]\ R \Leftrightarrow [\ T(\text{op})\ (T(\text{par\_list}))\ ]\ R$

MACHINE ***x_base***
SEES *SS*
SETS $t1\ ,\ \ldots\ ,\ tn$
CONSTANTS $IC\ ,\ I1\ ,\ldots\ ,\ In$
PROPERTIES $IC \subseteq SS \wedge I1 \in IC \rightarrow t1 \wedge \ldots \wedge In \in IC \rightarrow tn$
VARIABLES $C\ ,\ a1\ ,\ldots\ ,\ an$
INVARIANT $C \subseteq SS \wedge a1 \in C \rightarrow t1 \wedge \ldots \wedge an \in C \rightarrow tn$
INITIALISATION $C := IC \ \parallel\ a1 := I1 \ \parallel\ \ldots \ \parallel\ an := In$
OPERATIONS

$SKIP\_C \ \widehat{=}$ skip ;

$ADD\_C(\ vc\ ,\ v1\ ,\ldots,\ vn\ ) \ \widehat{=}$
PRE $vc \in SS \wedge vc \notin C \wedge v1 \in t1 \wedge \ldots \wedge vn \in tn$
THEN $C\ ,\ a1\ ,\ \ldots\ ,\ an :=$
$C \cup \{\ vc\ \}\ ,\ a1 \leftarrow\!\!+\ \{\ vc \mapsto v1\ \}\ ,\ \ldots\ ,\ an \leftarrow\!\!+\ \{\ vc \mapsto vn\ \}$
END ;

$REM\_C\ (\ vc\ ) \ \widehat{=}$
PRE $vc \in C$
THEN $C, a1, \ldots, an := C - \{\ vc\ \}, \{\ vc\ \} \lhd a1\ ,\ \ldots\ ,\ \{\ vc\ \} \lhd an$
END ;

$CHANGE\_C\_a1\ (\ vx\ ,\ va1\ ) \ \widehat{=}$
PRE $vx \in C \wedge va1 \in t1$
THEN $a1 := a1 \leftarrow\!\!+\ \{\ vx \mapsto va1\ \}$
END ;

...

$CHANGE\_C\_an\ (\ vx\ ,\ van\ ) \ \widehat{=}$
PRE $vx \in C \wedge van \in tn$
THEN $an := an \leftarrow\!\!+\ \{\ vx \mapsto van\ \}$
END

END

Figure 2: Prototypical Base Machine

Each class in a Structured Database Schema is translated into 3 abstract machines. For each class we give a *base machine* with class variables and base operations, a *class machine* which uses base operations to provide operations on the class, and a top-level machine which captures the class with its *is-a** relation

Base Machines. A *base machine* is constructed for each class, containing the class variables and attributes, its implicit constraints, and an AMN represen-

tation of the base operations. As in ASSO these implicit constraints are automatically preserved, the consistency of this machine does not need to be proved. By placing these operations into a separate machine we exploit the semi-hiding principle and do not allow the implicit constraints to be violated.

We denote this machine as: $C_{base} = T_{base}(C)$. This base machine for the general class C is given as follows. The given set S generates a stateless machine SS containing only the set S. The corresponding generic base machine is given in Figure 2. Thus each class has its own class set, attributes, invariants, initialisation and operations. A base machine models a class as a set representing class instances, with total functions for attributes. For each class, we declare a set of basic operations similar to those used in ASSO.

Theorem 1. *The translation T_{base} is sound.*

Proof. (Sketch). To show that this translation is sound, we prove that the translation of the class fragment formed by taking the instance and attribute sets of class C together with the base operations and the initialisation.

Clearly the variables and attributes are translated as in condition 1, and there are no explicit constraints in this fragment, so soundness condition 2 holds. For condition 3, consider the base operation $ADD\ C\ (x,\ v_1,\ \ldots,\ v_n)$.

$$T(ADD\ C\ (x,\ v_1,\ \ldots,\ v_n)) = ADD_C\ (x,\ v_1,\ \ldots,\ v_n)$$

as the translation on values is identity. We first show $[\ op\ (par\_list)\ ]\ R \Rightarrow [\ T(op)\ (T(par\_list)\ )\ ]\ R$ Assume, for predicate R the antecedent holds:

$$\begin{aligned}
&[ADD\ C\ (x,\ v_1,\ \ldots,\ v_n)]R \\
&\quad = \exists\, C', a'_1, \ldots, a'_n \cdot (R' \wedge PRE_C(op\ C\ (par_list)\ ) \wedge prd_C(op\ C\ (par_list))) \\
&\quad = \exists\, C', a'_1, \ldots, a'_n \cdot (R' \wedge PRE_C(op\ C\ (par_list)\ ) \wedge \\
&\qquad (C' = C \cup \{x\} \wedge (a'_1 = a_1 \lhd\!\!+ \{(x,\ v_1)\})) \wedge \ldots \wedge (a'_n = a_n \lhd\!\!+ \{(x,\ v_n)\})
\end{aligned}$$

Applying $\exists$-Elimination with the witnesses: $C' = C \cup \{x\}, a'_1 = a_1 \lhd\!\!+ \{(x, v_1)\}, \ldots, a'_n = a_n \lhd\!\!+ \{(x, v_n)\}$) results in:

$$\begin{aligned}
&(x \in S \wedge x \notin C \wedge v_1 \in t_1 \wedge \ldots \wedge v_n \in t_n) \wedge \\
&\qquad [C' := (C \cup \{x\}), a'_1 := a_1 \lhd\!\!+ \{(x, v_1)\}, \ldots, a'_n := a_n \lhd\!\!+ \{(x, v_n)\}]R'
\end{aligned}$$

Which is the same predicate as:

$$\begin{aligned}
&[ADD_C\ (x,\ v_1,\ \ldots,\ v_n)]R = (x \in S \wedge x \notin C \wedge v_1 \in t_1 \wedge \ldots v_n \in t_n) \wedge \\
&\qquad [C := (C \cup \{x\}),\ a_1 := a_1 \lhd\!\!+ \{(x, v_1)\}, \ldots, a_n := a_n \lhd\!\!+ \{(x, v_n)\}]R
\end{aligned}$$

The argument can be given in reverse to show $[\ op\ (par\_list)\ ]\ R \Leftarrow [\ T(op)\ (T(par\_list)\ )\ ]\ R$. A similar argument holds for the other base constructors. Thus a base machine is a sound translation of the base operations of a class.

Note that as a consequence of this proof, the base operations of machine C_{base} must preserve its invariant, which is the translation of *class-constraints*$_C$. The ASSO base operations have been shown to preserve this property, and thus so do the translated operations. There still remains the proof obligation that the initialisation establishes the class constraints (invariant) on $T(C)$.

Class Machines. The next stage is to **INCLUDE** base machines in a class machine for each *class*. The operations of this machine are directly translated from the corresponding ASSO operations. This machine also includes any explicit constraints on the class, generating the appropriate proof obligations. The generic class machine is outlined in Figure 3. The base operations of the class are not exported outside this machine, and thus are not required to satisfy the explicit invariants on the machine.

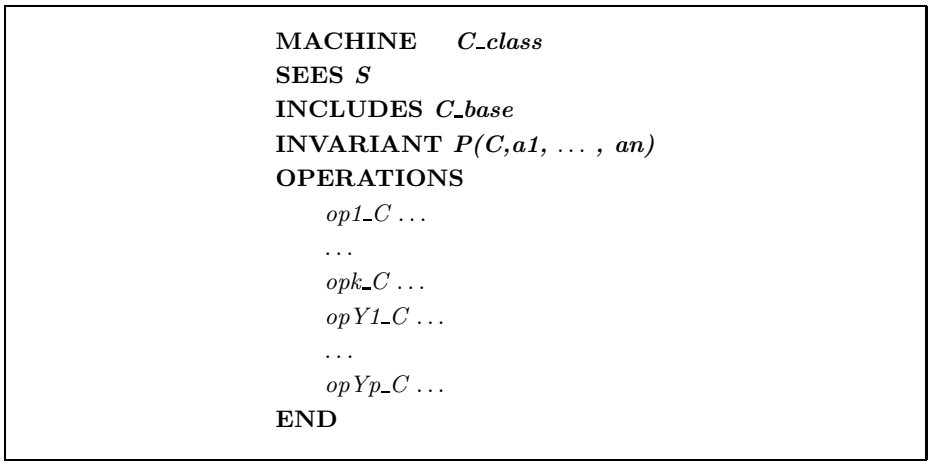

Figure 3: Prototypical Class Machine

A significant difference between the B method and ASSO is shown by the treatment of *implicitly* inherited operations [20]. In ASSO, inherited operations which are not redefined in the subclass are inherited and specialised via the *is-a** relationship. However, in B such operations have to be redefined *explicitly* within the class machine. Such specialisation can be systematically generated by a syntactic transformation on the operations of the superclass, renaming all instances of the identifiers of the superclass to that of the subclass. This is given by the transformation $E_{Y,X}$ on operations op_Y in the superclass Y:

$Y \longrightarrow X$ $\qquad$ $ADD_Y \longrightarrow SKIP_X$

$SKIP_Y \longrightarrow SKIP_X$ $\qquad$ $CHANGE_Y \longrightarrow SKIP_X$

$REM_Y \longrightarrow REM_X$ $\qquad$ $op_Y \longrightarrow op_X$

and recursively through the operation constructors.

Effectively, we only specialise explicitly those ASSO operations which delete objects as part of their action. Operations which insert are already explicitly specialised as a part of the definition of well formed structured database schemas in ASSO. Operations which modify attributes of the superclass are not well-defined in the subclass considered in isolation and can be regarded as *skip*. In the generic schema in Figure 3, the operations *opY1_X*, ..., *opYp_X* represent the explicitly inherited operations. In Figure 1 above, *del.employee* would have to be explicitly defined in the class machine for *employee*.

Thus the translation of the structured database schema into B is not a composition of translations of database schemas. Each class machine is the result of a top-down analysis of the whole hierarchy to generate specialisations of inherited operations. This is reflected in the formal definition of the translation. The soundness proof is direct.

Definition 3. *The class translation is denoted* T_{class}. *If* C is-a* C', *the operations of* $\mathrm{T}_{class}(C)$ *are* $\mathrm{T}_{class}(\mathrm{OP}_C) \cup \mathrm{E}_{C',C}(\mathrm{T}_{class}(\mathrm{OP}'_C))$, *where* OP_C *are the operations of* C.

Theorem 2. T_{class} *is sound.*

Proof. (Sketch). The class and attribute variables are included via the T_{base} relation, which is sound, and the explicit constraint is directly included in $T_{class}(C)$. The explicitly specialised operations of C, *op* (*par_list*) are directly translated into $T_{class}(C)$, and using the soundness of the T_{base} relation, and the equivalent semantics of generalised substitution constructors in ASSO and B, we must therefore have that:

$$[op\ (par\_list)]\ R \Leftrightarrow [\,T_{class}(op)(\,T_{class}(par\_list)\ )\,]\ R$$

If C is a root class we are finished. If C *is-a** C', then by definition of *is-a** C has implicitly inherited and specialised the C' operation *op* (*par_list*) *op (par_list)*. By the above definition of $T_{class}(C)$, $E_{C',C}(\,T_{class}(op(par\_list)\ )$ is an operation on $T_{class}(C)$, and, via a recursive analysis of the constructs, the property holds.

The Top-Level Machine. For a complete analysis of class C in B, for each *class*, a top-level machine can be constructed, with the implicitly composed operations given explicitly. These operations are formed by the placing the inherited operation in parallel with the operation of that class, together with the implicit constraints generated via the *is-a** relation. We give the formal definition of the translation, $T_{top}(C)$.

Definition 4. *If* C *is a root class, then* $T_{top}(C) = T_{class}(C)$. *If* C is-a* C', *then* $T_{top}(C)$ *is a B machine formed as follows:*

1. **INCLUDE** *the class machines* $\mathrm{T}_{class}(C)$*, and* $\mathrm{T}_{class}(C')$*.*
2. *Add to the* **INVARIANT** *the implicit* is-a* *constraint.*
3. *For each operation* $\mathrm{op}_C(x, u_1, \ldots, u_j\ )$ *in* $T_{class}(C)$ *and* $\mathrm{op}_{C'}(x, v_1, \ldots, v_k\ )$ *in* $T_{class}(C')$*, where* x *is a common instance identifier variable, form the combined operation:*

$$\mathrm{op}(x, u_1, \ldots, u_j, v_1, \ldots, v_k\ ) = \mathrm{op}_C(x, u_1, \ldots, u_j\ )\ \|\ \mathrm{op}_{C'}(x, v_1, \ldots, v_k)$$

Theorem 3. *The translation* T_{top} *is sound.*

Proof. (Sketch). If C is a root class, then $T_{top}(C) = T_{class}(C)$, which is sound by the above theorem. If C *is-a** C', then we need only show part 3 of the definition of soundness. We can assume that the machines $T_{class}(C)$, and $T_{class}(C')$ are sound. Thus we have for each operation its C and its correspondent in C':

$$[op(par\_list)]\ R \Leftrightarrow [T_{class}(op)(T_{class}(par\_list))]\ R$$
$$[op'(par\_list)]\ R \Leftrightarrow [T_{class}(op')(T_{class}(par\_list))]\ R$$

Thus we have (omitting *par_list* for clarity)

$$[op]\ R \wedge [op']\ R \Leftrightarrow [T_{class}(op)]\ R \wedge [T_{class}(op')]\ R$$

As C and C' have independent state variables, from [2], §7.1.3, we have that:

$$[op]\ R \wedge [op']\ R \Leftrightarrow [op\ \ \|\ \ op')]\ R$$
$$[T_{class}(op)]\ R \wedge [T_{class}(op')]\ R \Leftrightarrow [T_{class}(op)\ \ \|\ \ T_{class}(op')]\ R$$

And we are done.

As the operations have been proven to be correct, then these operations do not need to be reproven to be consistent by virtue of the structure of ASSO, and it is unnecessary to generate a top-level machine to establish consistency against the inherent constraints of the *is-a** relation.

4 Refinement in ASSO

ASSO has two complementary forms of refinement:

– *Behavioural Refinement.* Operations are refined towards implementation by reducing non-determinism, weakening preconditions, and reducing partiality of operations. Such a refinement between classes C_1, C_2 is denoted: $C_1 \Leftarrow C_2$.

– *Data Refinement.* ASSO data refinement consists of a stepwise decomposition of classes organised into a specialisation hierarchy, representing a conceptual schema. A set of smaller specialisation hierarchies is generated step by step until a set of disjoint classes is obtained, which defines the ASSO logical schema having separate classes for each intersection class. The ASSO data refinement uses the Partitioning Method, a recursive algorithm working on graphs. Such a refinement between classes C_1, C_2 is denoted: $C_1 \equiv C_2$.

To verify an ASSO refinement, the relationship which links the corresponding consistency proof obligations of any two *behaviourally* refined schemas must be logical implication. The relationship which links corresponding consistency proof obligations before and after partition is logical equivalence. The partitioning algorithm ensures that this holds without proof.

5 Behavioural Refinement

Behavioural refinement modifies operations, whilst preserving the consistency of the operations. Thus we can define a behavioural refinement as follows.

Definition 5. *Given ASSO operations* op_1 *and* op_2 *where* op_1 *terminates and satisifies the constraints* P *(that is* $P \Rightarrow [\ \mathrm{op}_1\ ]P$*) and if* op_2 *terminates and satisfies* P' *such that:* $\mathrm{trm}(\mathrm{op}_1) \subseteq \mathrm{trm}(\mathrm{op}_2) \wedge P \Rightarrow P'$*, then we can say that* op_2 *is a refinement of* op_1 *with respect to* P*, written* $\mathrm{op}_1 \Leftarrow \mathrm{op}_2$.

We can then use this definition to define refinement of classes.

Definition 6. *Given classes* C_1 *and* C_2 *which have the same instances, attributes, application constraints and operation names, then*

$$C_1 \Leftarrow C_2 \text{ if and only if } \forall\ \mathrm{op}_{C_1} \in C_1 \cdot \mathrm{op}_{C_1} \Leftarrow \mathrm{op}_{C_2}$$

with respect to the implicit and explicit constraints of C_1.

Thus we can prove the refinement of a classes by considering the refinement of each of its operations in turn (including the implicitly specialised ones). Note that the similar proposition for B does not hold.

The forms of behavioural refinement above satisfy this definition of refinement.

Theorem 4. *Weakening the precondition, reducing partiality, and reducing nondeterminism are all refinements.*

Proof. We assume that we have classes C_1, C_2 with implicit and explicit constraints on C_1, P. Then a given op_{C_1} is transformed into op_{C_2} and we show that $op_{C_1} \Leftarrow op_{C_2}$.

Weakening the precondition. If $PRE(op_{C_1}) \Rightarrow PRE(op_{C_2})$, then $pre(op_{C_1}) \subseteq pre(op_{C_2})$. So by definition, $trm(op_{C_1}) \subseteq trm(op_{C_2})$. As the substitution of the operation has not changed, $rel(op_{C_1}) = \{trm(op_{C_1})\} \lhd rel(op_{C_2})$ (where $\lhd$ is domain restriction), P is preserved by op_{C_2} and hence $trm(op_{C_1}) \subseteq trm(op_{C_2} \wedge P \Rightarrow P'$ and thus it is a refinement.

Reducing partiality. If $trm(op_{C_1}) \subseteq trm(op_{C_2})$, again the operation's substitution has not changed, so $rel(op_{C_1}) = \{trm(op_{C_1})\} \lhd rel(op_{C_2})$ and P is preserved by op_{C_2}, hence: $trm(op_{C_1}) \subseteq trm(op_{C_2}) \wedge P \Rightarrow P'$ and it is a refinement.

Reducing non-determinism. In this case there two properties: $pre(op_{C_1}) = pre(op_{C_2})$, hence: $trm(op_{C_1}) = trm(op_{C_2})$, and $rel(op_{C_2}) \subseteq rel(op_{C_1})$ that is the before-after relation becomes more restricted. Thus for any $(s, s') \in rel(op_{C_2})$, $(s, s') \in rel(op_{C_1})$, so P holds and we have a refinement.

class *student* **is-a*** *person* **with** (*matriculation*:N
init.student() $\widehat{=}$ *student*, *matriculation* := {},{}
new.student(*pers*) $\widehat{=}$
PRE *pers* $\in$ *PERSON* $-$ *student*
THEN *ADD student*(*pers*, max(ran(*matriculation*))+ 1)
END)

Figure 4: A Behavioural Refinement of *new. student*

ASSO refinement is *compositional*. That is, in a Structured Database Schema, if a class is refined, then the whole schema is refined. This is captured in the following theorem.

Theorem 5. *If C_1 and C_1 are classes, and the Structured Database Schema C_1* is-a* *C_2 is constructed, where there are no explicit application constraints between the classes, then the following two properties hold:*

i). If $C_1 \Leftarrow C_1'$ then C_1 is-a* *$C_2 \Leftarrow C_1'$* is-a* *C_2.*
ii). If $C_2 \Leftarrow C_2'$ then C_1 is-a* *$C_2 \Leftarrow C_1$* is-a* *C_2'.*

Thus we can say that is-a* *is* monotonic *with respect to refinement.*

Proof. (Sketch): i). If $C_1 \Leftarrow C_1'$ then for any operation $op = op_{C_1} \parallel op_{C_2}$. Therefore if $op_{C_1} \Leftarrow op_{C_1'}$, then as S and R have independent variables, we have $op_{C_1} \parallel op_{C_2} \Leftarrow op_{C_1'} \parallel op_{C_2}$. A similar argument holds for ii).

Thus we can perform *modular refinement* on ASSO schemas. This too is in contrast to B where the compositional structures are not in general monotonic through refinement.

structured database schema
database1
class *person−employee* **of** *PERSON* **with** (*income*:*N*
constraints ∀ ***p*** **(*p*∈*person−employee* ⇒ *income*(*p*)≥ 1000)**
init.person−employee() = *person*, *income* := {},{}
new.person(*pers*, *i*) ≙
PRE *pers* ∈ *PERSON* − (*person−employee*) ∧ *i* ≥ 1000
THEN
ADD person−employee(*pers*, *i*)
END ;
del.person−employee(*person*) ≙
PRE *pers* ∈ *person−employee*
THEN *REM person−employee*(*pers*)
END)
class *student* **is-a*** *person−employee* **with** (*matriculation*:*N*
constraints ∀ ***s1, s2*** **(*s1*∈*student* ∧ *s2*∈*student* ∧ *s*1 ≠ *s*2 ⇒
matriculation(*s1*)≠*matriculation*(*s2*))
init.student() ≙ *student*, *matriculation* := {},{}
new.student(*pers*) ≙
PRE *pers* ∈ *PERSON* − *student*
THEN
ANY *m* **WHERE** *m* ∈ *N* ∧ *m* ∉ ran(*matriculation*)
THEN *ADD student*(*pers*, *m*)
END
END)

Figure 5: The first SDBS generated by a partitioning step.

The SDBS in Figure 1 defines the non-deterministic operation *new.student*; it selects any new value for the matriculation number which has not been used before. A more determined version of *new.student* is given in Figure 4 (constraint omitted). In this case, the value of the new *matriculation* is one greater than the maximum of all the existing matriculation numbers. This new class *student1* is a refinement of *student*, and thus by Theorem 5, the structured database schema formed by replacing *student* with *student1* in *database* is also a refinement.

5.1 An Example of Data Refinement

The Partitioning algorithm is complex, and its details are omitted here. To demonstrate its effect, we apply the first step of the partition method to the example in Figure 1.

structured database schema
database2
class *person•employee* **of** *PERSON* **with** (*income*:*N*
constraints $\forall$ ***p*** **(*p*∈*person•employee* ⇒ *income*(*p*)≥ 1000)**
init.person•employee() = *person*, *income* := {},{}
new.person(*pers*, *i*) $\widehat{=}$
PRE *pers* ∈ *PERSON* − (*person•employee*) ∧ *i* ≥ 1000
THEN *ADD person•employee*(*pers*, *i*)
END ;
del.person•employee(*person*) $\widehat{=}$
PRE *pers* ∈ *person•employee*
THEN *REM person•employee*(*pers*)
END)
class *student* **is-a*** *person•employee* **with** (*matriculation*:*N*
constraints $\forall$ ***s1, s2*** **(*s1*∈*student* ∧ *s2*∈*student* ∧ *s*1 ≠ *s*2 ⇒**
matriculation(*s1*)≠*matriculation*(*s2*))
init.student() $\widehat{=}$ *student*, *matriculation* := {},{}
new.student(*pers*) $\widehat{=}$
PRE *pers* ∈ *PERSON* − *student*
THEN
ANY *m* **WHERE** *m* ∈ *N* ∧ *m* ∉ ran(*matriculation*)
THEN *ADD student*(*pers*, *m*)
END
END)

Figure 6: The second SDBS generated by a partitioning step.

The initial top-level class *person* has been decomposed into two: the class disjoint from the class *employee*, producing the SDBS given in Figure 5, and the intersection class with the class *employee* generating the structured database schema given in Figure 6. In this latter, the new class intialisation and constraints of *employee*, and the operations on this class are parallel compositions of the corresponding operations on the classes *person* and *employee*. Each SDBS also takes a copy of the subclass *student* implicitly splitting this class between the two partitions of *person*. This partitioning can be continued to generate a logical schema of disjoint classes.

5.2 The Relationship between the Types of ASSO Refinement

Behavioural refinement and partitioning are not independen t in ASSO; partitioning also modifies operations. Consider a class A with an operation op on one class instance which we represent as a function on its state $op : A \rightarrow A'$. Then if A is partitioned with respect to a class B, it is divided into two sections: $A \bullet B$ and $A - B$. The operation is also partitioned: $op_{A \bullet B} : A \bullet B \rightarrow (A \bullet B)'$, and $op_{A-B} : A - B \rightarrow (A - B)'$. These two new operations can be combined into a new one which refines the original:

$$op \Leftarrow op_{A \bullet B} \parallel op_{A-B}$$

This holds as these operations on the class A do not depend on the values of B, which may or may not be present in the new classes. Thus partitioning is well behaved with respect to behavioural refinement for operations that only access one class instance. However, if more than one class instance is referenced, problems can occur. Consider the following example. Within the class A we define the following ASSO operation:

remove_either$(p, q) =$
PRE $p \in A \wedge q \in A$
THEN CHOOSE *REM A p*
OR *REM A q*
END
END

If A is partitioned with respect to class B, the following two operations result:

remove_either$_{A \bullet B}(p, q) =$
PRE $p \in A \bullet B \wedge q \in A \bullet B$
THEN CHOOSE *REM* $A \bullet B$ *p*
OR *REM* $A \bullet B$ *q*
END
END

remove_either$_{A-B}(p, q) =$
PRE $p \in A - B \wedge q \in A - B$
THEN CHOOSE *REM* $A - B$ *p*
OR *REM* $A - B$ *q*
END
END

Clearly this does not partition the operation as not all the of the domain has been covered. The parameters could also satisfy the preconditions: $p \in A \bullet B \wedge q \in A - B$ or $p \in A - B \wedge q \in A \bullet B$. Thus there are two other operations which should be generated to produce a valid partition of the operation.

remove_either$_1(p, q)$ =
PRE $p \in A \bullet B \wedge q \in A - B$
THEN CHOOSE *REM* $A \bullet B$ p
OR *REM* $A - B$ q
END
END

remove_either$_2(p, q)$ =
PRE $p \in A - B \wedge q \in A \bullet B$
THEN CHOOSE *REM* $A - B$ p
OR *REM* $A \bullet B$ q
END
END

However, it is unclear to which class these should belong. Clearly they cannot belong in either class as this would violate the principle that a class's operations only access the variables of that class. They should belong in a class encompassing both $A \bullet B$ and $A - B$; however, adding this class contradicts the goal of partitioning of separating out all classes into independent classes.

This example demonstrates that partitioning is not in itself a valid *behavioural* refinement. However, the following observation offers a solution to this problem. We observe that both *remove_either*$_{A \bullet B}$ and *remove_either*$_{A-B}$ can be behaviourally refined further, first with a step of reduction of non-determinism, and then with a weakening of precondition, resulting in the operations:

remove_either$'_{A \bullet B}(p, q)$ =
PRE $p \in A \bullet B$
THEN *REM* $A \bullet B$ p
END

remove_either$'_{A-B}(p, q)$ =
PRE $p \in A - B$
THEN *REM* $A - B$ p
END

Now we have that

$$\textit{remove\_either}(p,q) \Leftarrow \textit{remove\_either}'_{A \bullet B}(p,q) \parallel \textit{remove\_either}'_{A-B}(p,q)$$

T hus we have restored the refinement. This can be done in general due to the restricted nature of ASSO operation constructors and refinement.

However, we note that in general, even if partitioning is a behavioural refinement, if it is done before other behavioural refinement steps, then the number of behavioural refinement steps increases. A class divides into one class which intersects with a given class, and another which intersects with that class' complement. An operation also divides into two, and will require two behavioural steps to commute. Thus for efficiency as well as to avoid the above problem, ASSO defines a process where behavioural refinement is undertaken first, and partitioning second.

5.3 The Relationship between ASSO and B Refinement

B refinement supports development into code. Like behavioural refinement in ASSO this allows the weakening of preconditions, reduction of non-determinism, and partiality of operations. B also allows programming like constructs such as loops; these are disallowed in ASSO which remains a high-level design language. M is refined into machine N if they are related via a *coupling condition* in the invariant of N, and each operation in M has a counterpart in N. Proof obligations verify refinement; that of most interest establishes that the behaviour of the refined operation respects that of the original, within the original's precondition. Thus if in machine M with invariant I has operation $op =$ **PRE** P **THEN** S **END** and in machine N, with invariant J which includes a coupling condition, where $op =$ **PRE** Q **THEN** T **END**, the proof obligation:

$$P \wedge I \Rightarrow Q \wedge [T]\neg[S]\neg J$$

establishes the refinement.

To use B's refinement proofs to support ASSO, we need to show that an ASSO refinement is a B refinement. Behavioural refinement for ASSO and B are clearly similar. There is no change in data model, although the values held by attributes may differ. Thus the coupling condition is that the class members are the same before and after the refinement.

An ASSO behavioural refinement is a B refinement.

Theorem 6. *Given classes C_1, C_2 and $C_1 \Leftarrow C_2$ in ASSO, then $T(C_1) \Leftarrow T(C_2)$ in B*

Proof. If we have $C_1 \Leftarrow C_2$ in ASSO, then we have that pairwise $op_{C_1} \Leftarrow op_{C_2}$. We show that $T(op_{C_1}) \Leftarrow T(op_{C_2})$ in B. A definition of refinement of operations in B is given ([2], §11.1.2) as:

$$op_1 \Leftarrow op_2 \Leftrightarrow pre(op_1) \subseteq pre(op_2) \wedge rel(op_2) \subseteq rel(op_1)$$

Now we have seen from the proof of the previous theorem that this holds (modulo termination range restrictions to termination sets). Thus we have that an ASSO refinement can be translated into a B refinement.

A step of data refinement partitions each class into two, and also partitions operations. The coupling constraint in thus that the union of the two partitions after refinement is the same as the whole class before, i.e. if class C is being partitioned away from class D, then the coupling condition is: $C = (C \bullet D) \cup (C - D)$. If the union of partitioned operations cover the original operation, in the case where only one member of the class is accessed in an operation, then the parallel composition of the two parts of the partitioned operation constitute a B

refinement of the original operation. Since the partitioning algorithm preserves the equivalence of the models, refinement proofs in this case are unnecessary.

Thus in both cases ASSO refinement is B refinement, and Theorem 7 follows.

Theorem 7. *If SDBS S, represented by B Abstract Machine M, is refined into a set of Structured Database Schema R , and R is represented by B Abstract Machine N, then N is a B-refinement of M.*

As behavioural refinement of operations is a B refinement, it can be validated using B tools, through the translation of ASSO schema. Translating an ASSO refinement into B exploits some properties of the two languages. As in a B refinement the abstract invariant has to be respected by the refining machine, it does not need to be repeated. Also we only define the refinement relation between the top-level machine of each class. For example, the systematic translation of the behavioural refinement in Figure 4 into B notation and proof using the B-Toolkit [4] confirms that both *student1* is a refinement of *student*, and also that the structured database schema *database1*, formed by replacing *student* by *student1* in *database* is indeed a refinement.

6 Conclusions

ASSO is a formal methodology which has advantages for the design of databases compared to the other formal methods currently used in industrial applications. The *is-a** relationship between structured database schema and the Partitioning Method are original aspects of both formal methods and the database areas, providing a higher-level of conceptual design to the database developer, exploiting the results in conceptual schemas dating back from the late 70s.

The results in this paper give three things:

i. a firm foundation for the development of databases within ASSO in terms of the semantics of B;
ii. an established proof theory for demonstrating the consistency and correctness of ASSO models;
iii. tools developed to support B which can be used to support ASSO.

The restricted subset of B's features used by ASSO allow a relationship to be established in a straighforward manner.

ASSO should also be compared with approaches to Object-Oriented design within B, for example [9, 10, 12, 13, 22]. ASSO offers a different approach which in specialisation offers different properties in inheritance of operations. Also, by considering object-oriented databases with structuring and inheritence, we go further than the relational approach of specifiying databases in B in [21].

To progress further with ASSO, including using it on practical problems, it is proposed to design tools which aid the user to construct an ASSO design, control its structure, to prove properties, and to move towards an implementation. The ASSO tools can be based on a part of the B Toolkit, together with new tools, such as those supporting partitioning. An initial design for these tools, which integrate with the B-Toolkit, is discussed in [18].

Acknowledgements

This work has been supported by the CNR short-term mobility programme, 1998. The authors would like to thank the chairman of the B track of FM99 and the anonymous referees for their useful comments and suggestions.

References

[1] S Abiteboul, R Hull and V Vianu: *Foundations of Databases*, Addison-Wesley, 1995.
[2] J-R Abrial: *The B-Book: Assigning Programs to Meaning*, Cambridge University Press, 1996.
[3] R Andolina and E. Locuratolo: *ASSO: behavioral specialisation modelling.* In: Jaakkola H (ed) Information Modelling and Knowledge Bases VIII, IOS Press, 1997.
[4] B-Core (UK) Ltd. *The B-Toolkit Welcome page* URL <http://www.b-core.com>
[5] N. Bhalla: *Object-oriented data models: a perspective and comparative review,* Journal of Information Science, 1991.
[6] D. Castelli, and E. Locuratolo: *A Formal Notation for Database Conceptual Schema Specifications.* Information Modelling and Knowledge Bases VI, Jaakkola, H. (ed), IOS Press, 1994.
[7] D. Castelli and E. Locuratolo: *ASSO - A formal database design methodology* In Jaakkola H (ed) Information Modelling and Knowledge Bases VI, IOS Press, 1995.
[8] E W Dijkstra., and S Scholten: *Predicate Calculus and Program Semantics,* Springer-Verlag, 1990.
[9] P Facon, R Laleau and H P Nguyen: *Mapping object diagrams into B specifications.* In Bryant A, Semmens L T (eds) proceedings of the Methods Integration Workshop, Leeds, UK, Springer-Verlag, 1996
[10] P Facon, R Laleau and H P Nguyen: *Dèrivation de spècification formelles B à partir de spècifications semi-formelles de systëmes d'information.* Habrias H (ed) proceedings of the 1st Conference on the B Method, Nantes, France, 1996.
[11] R. Hull and R. King: *Semantic database modeling: survey, application, and research issues.* ACM Computing Surveys, 19(**3**) 1987
[12] K Lano and H Haughton: *Improving the process of system specification and refinement in B.* In: Till D (ed) proceedings of the 6th Refinement Workshop, Workshops in Computing, Springer-Verlag, 1994.
[13] K Lano: *The B Language and Method. A Guide to Practical Formal Development.* Springer-Verlag, 1996.

[14] E. Locuratolo: *Evolution as a formal database design methodology.* In Proceedings of symposium on Software Technology (SoST'97), Buenos Aires, 1997.
[15] E. Locuratolo and R. Rabitti F: *Conceptual classes and system classes in object databases.* Acta Informatica, 35(**3**):181-210, 1998.
[16] E.Locuratolo:*Database reengineering as modelling* IEI report B4-43, 1998.
[17] E. Locuratolo, and B.M. Matthews: *On the relationship between ASSO and B* In proceedings of the 8th European-Japanese conference on Information Modelling and Knowledge Bases, Vammala, Finland, May 1998.
[18] E. Locuratolo, and B.M. Matthews: *ASSO: A formal methodology of Conceptual Database Design* ERCIM Workshop on Formal Methods for Industrial Critical Systems, 1999.
[19] B.M. Matthews, B Ritchie, and J.C. Bicarregui: *Synthesising structure from flat specifications.* Proceedings of the B Conference, Montpelier, France, April 1998.
[20] B.M. Matthews and E. Locuratolo: *Translating Structured Database Schemas into Abstract Machines.* In proceedings of the 2nd Irish Workshop on Formal Methods, Cork, Ireland, 1998.
[21] K-D Schewe, J W Schmidt and I Wetzel: *Specification and refinement in an integrated database application environment.* Prehn S, Toetenel W J (eds) proceedings of VDM'91, 4th Int. Symp. of VDM Europe, Springer-Verlag, 1991
[22] R Shore: *An object-oriented approach to B.* Habrias H (ed) proceedings of the 1st Conference on the B Method, Nantes, France, 1996.