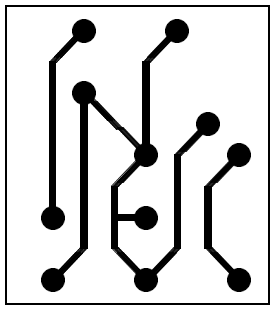
Instituto de Engenharia de Sistemas e Computadores

Departamento de Informática da U.Minho



2361 - Braga

Universidade do Minho
INESC

Campus de Gualtar,
4700 Braga
Portugal

Tel: +351-53-604470
Fax: +351-53-612954

Project C.19. Sviluppo di Metodologie, Sistemi e Servizi Innovativi in Rete

Oliveira J.N., Cruz A.M.

Formal Calculi Applied to Software Component Knowledge Elicitation

Formal Calculi Applied to Software Component Knowledge Elicitation

by

Oliveira J.N., Cruz A.M.
DI/INESC
Universidade do Minho,
Gualtar,
4700 Braga
Portugal

December 1993

Abstract

This report illustrates the application of formal calculi in software component knowledge elicitation and classification. In particular, it shows how the SETS program calculus [9, 11] can be useful not only to refine software but also to extract "knowledge" from existing software artifacts; to reason about such formal knowledge descriptions, *e.g.* by comparing them, calculating their "difference", *etc.*; and to effectively classify software components according to an underlying formal classification scheme.

This approach is opposed to conventional 'ad hoc' classification schemes not only because of its formal basis but also in defining a structural *similarity* relation between software components which supports their analogical reuse across unrelated domain areas.

Finally, the strategy is specifically applied to ERA-diagram knowledge elicitation, reasoning and classification.

Contents

1 Introduction **4**
1.1 Report Structure . . . 5

2 On Conventional Software Comparison Metrics **6**
2.1 The Keyword-based Approach . . . 7
2.2 The Attributive Approach . . . 8
2.3 The Faceted Perspective . . . 9

3 An Alternative Formal Comparison Strategy **9**
3.1 What "is" a Software Component? . . . 10
3.2 The Classify-by-Data Perspective . . . 12
3.2.1 Syntatical Approach . . . 12
3.2.2 Semantic Approach . . . 15
3.3 Additional Structural Relationships among Software Components . . . 18
3.3.1 Desired Properties . . . 19

4 Application to ERA-diagram Classification **19**
4.1 First Step: A SETS Semantics for ERA-diagrams . . . 19
4.1.1 Sub-types . . . 21
4.1.2 '1:N' Relationships . . . 22
4.1.3 '1:1' Relationships . . . 25
4.1.4 'N:M' Relationships . . . 26
4.2 Second Step: Structure Abstraction in SETS . . . 29

5 An Example **30**

6 Conclusions and Further Work **33**

References **34**

A An Overview of the SETS Notation **35**

B A Summary of the SETS Calculus **38**
B.1 Equational Calculus Fragment . . . 38
B.2 Inequational Calculus Fragment . . . 38
B.3 Abstraction Functions . . . 39
B.4 Invariant Predicates . . . 39

C An ERA-diagram Semantics Cookbook **39**
C.1 Entities . . . 39
C.2 Relationships . . . 40

D A Tool for ERA-diagrams Conversion to SETS **43**

Context

This report presents a R&D effort to apply a formal calculus to the comparison and classification of software knowledge descriptions, as scheduled for workpackage WP2D of the collaboration offer by INESC to research project C.1.9. SVILUPPO DI METODOLOGIE, SISTEMI E SERVIZI INNOVATIVI IN RETE, in collaboration with OIS RICERCA S.c.p.A., Via Alberotanza 5, 70125 Bari, Italy.

Acknowledgements

INESC Group 2361 wish to thank Ing. Luciano Signorini (OIS RICERCA) for the proposal of the consultancy programme which encompasses this report. They also wish to thank the Departamento de Informática of the University of Minho for the permission to access to its library of workstudy-in-industry reports, making it possible to analyse and classify about twenty different information systems, as reported in this document.

1 Introduction

"Pen and pencil" development of software in the past misled people to think that program design would dispense with specific development tools. Software "in the small" appeared to be an attractive, cheap technology. However, lack of technological sophistication "in the large" has very often frustrated such expectations.

A basic problem with software development is the fact that code is hardly *re-used*, that is, it is difficult to produce a large software system by *composition* of smaller, readily available, software components. This has an obvious impact on increasing software production costs.

Software reusability is hindered by several reasons, some of which are even non-technical in nature [1]. Relatively recent interest on effectively promoting *reusability* in CASE technology tends to invert this situation. However, most CASE systems are oriented towards *production management* (versioning, documentation, team management *etc.*) rather than to the end *product*. This may be so because their designers couldn't probably understand what a software product "really is". Because software modules are regarded as mere text-files, their essence is never recorded. For many programmers, software re-use simply means "do not start from scratch: edit an existing source file". Software component repositories are built but either are not search effective or record trivial information (or even remain empty).

In order for existing software to be re-used in a new design one needs to be able to select, by comparison and ranking, the available software packages which better "approximate" the new problem. Thus re-usability brings about two basic problems: *comparison* and *classification* of software components. In fact, the (eventually) automatic *retrieval* of components from a *software repository* requires these to be "classified" and there is a need for what seems to be common practice elsewhere (*cf. e.g.* hardware, library systems, biology, *etc.*):

- a semantics-sensitive comparison procedure;
- a hierarchical classification schema

paving the way to the

- automatic (semantics-driven) component retrieval based on the adopted *classification schema* (*cf.* the ubiquitous "A is a B" relationship);

[1] See an interesting account of "sociological taboos" against reusability in *Social Aspects of Reusability.*, by Kruzela I., Telelogic, Sweden, 1989.

- definition of subsidiary relationships among components (*e.g.* a notion of *implementation*, "A implements B");

enabling effective re-use of machine code. The problem thus resides in a simple question: *. . . what is the "semantics" of a software component* upon which to base the whole thing? Or, put in another way: *. . . what is the best framework for capturing the "knowledge" implicit in 'software' packages?*

It is common practice nowadays to regard software modules as *objects* characterized by *attributes*, suitably framed in a user-defined class-system and supported by a given object-oriented programming environment.

It is the authors belief that attributive comparison is too general a framework for software classification — a topic which will be detailed in the sequel. Furthermore, reusability cannot be achieved by simple *ad hoc* means, even if addressed from a novel, object-oriented perspective. The basic question above — "what is a software component?" — has to do with software specification, and can only be properly answered in a *formal* context. This is so because some kind of mathematical "metrics" will be eventually required if the basic operations over a software component repository, namely *classification*, *comparison*, *retrieval* and *modification*, are to become fully automated.

INESC R&D group 2361 has paid some attention to the development of a formal notation — SETS [9, 11, 12] — able to support formal metrics for component classification based on its associated *calculus*. This report puts forward a strategy for building software component repositories whose essential feature is to characterize every software component by its formal specification, understood as a piece of data (internal state) and associated basic operations (actions) which are described and reasoned about using SETS[2].

SETS can be used not only to classify components (in a "classify-by-data" style, as is described in the sequel) but also to calculate component implementations or to compare them (by calculating their "difference"). Thus, architectural relationships in the repository such as *is-a*, *is-used-by*, *is-implementation-of*, *is-special-case-of* may be formally decided rather than arbitrarily fixed by the users intuition.

1.1 Report Structure

The SETS [11] notation and calculus will be extensively used in this report. Some understanding of the basic SETS constructs is required. The basic notation is skimmed through in appendix A (for further details see *e.g.* [10, 11]). Appendix B presents a fragment of the SETS calculus which is relevant with respect to the reasoning steps presented later on.

Section 2 overviews what we regard as "conventional" approaches to characterizing, comparing and classifying software. Section 3 proceeds to presenting an alternative formal characterization of software components based on the SETS calculus. In section 3.2 a SETS-calculus based comparison procedure is proposed for such components and a classification schema is built on top of such procedure. In section 3.3 other structural relationships typical of software component repositories are formally characterized using SETS. Finally, the proposed strategy is applied to classifying ERA-diagrams by reversing such diagrams to the SETS notation (section 4), and a brief illustration is given in section 5.

Appendix C presents a "cookbook" semantics for ERA-diagrams. Appendix D presents a tentative WINDOWS tool for assisting ERA-diagram to SETS conversion.

[2] A software component may thus be regarded as a *model* such as in the VDM methodology [7] or the CAMILA's notation [10]. The latter notation can also express the specification of a component as the functional *composition* of already available sub-components.

2 On Conventional Software Comparison Metrics

The reuse based approach to software development is today understood as a form of *analogical* problem solving strategy whereby the software engineer attempts to transfer solutions from one system (or problem) to another [20]. The basic question is how to define a quantitative *similarity* relation between software artifacts so as to promote their analogical reuse. Similarity can be described as a relation determined by some flexible *comparison* between the distinct constituents of two entities. From a quantitative viewpoint, comparison is normally interpreted as a measure of *closeness* in some abstract space.

This section presents a general framework amenable to express reuse-metrics at software component level. Acronym SC (for "software component") will be generically used to denote a given domain adopted for software knowledge description, to be suitably instantiated in later sections.

Comparison in mathematics is normally expressed by an ordering on the specific domain of reasoning. One may expect that some kind of computable "ordering" can be found on software components. But, due to software complexity, one should not expect much further than the following facts:

- Such an ordering will most probably be a *partial ordering*, because not every two components will be strictly comparable.
- In the situation where two given components are not comparable, one might expect that a *least upper-bound* component always exists which is the less abstract object which contains the "common information" of (*i.e.* which generalizes) the given objects.

Therefore, the discussion around software-comparison can be expressed in terms of a function

$$comp : SC \times SC \longrightarrow SC$$

which, given two software components (SCs), "computes" their *least upper-bound* SC.

Before going into details, one should assert some desirable properties of any candidate $comp$ function, the cornerstone of every comparison algebra:

- the comparison of an SC with itself should yield itself, that is,

$$comp(x, x) = x \tag{1}$$

- SC-comparison should be commutative, that is,

$$comp(x, y) = comp(y, x) \tag{2}$$

- SC-comparison should be associative, that is,

$$comp(x, comp(y, z)) = comp(comp(x, y), z) \tag{3}$$

- there must exist a "most abstract" SC, $\top$, which generalizes any other SC, that is,

$$comp(\top, x) = comp(x, \top) = \top \tag{4}$$

In this way, SCs will be assembled in a *specialize/generalize* semi-lattice with $\top$ at the top, a known framework for representing re-use relationships [13]. One will say that $SC\ x$ *specializes* $SC\ y$ (= $SC\ y$ *generalizes* $SC\ x$) wherever

$$comp(x, y) = y$$

holds, and will write

$$x \leq y$$

to denote such a specialization order.

We are ready to proceed to more specific characterizations of SC comparison.

2.1 The Keyword-based Approach

This approach is borrowed from classical library theory [16]. Software components are abstracted by sets of keywords capturing their most relevant features. We may thus write, in this case:

$$SC = 2^{Keyword}$$

Comparing two SCs x and y means to obtain a "measure" of what they have in common. In this case, one intuitively writes:

$$comp(x, y) = x \cap y$$

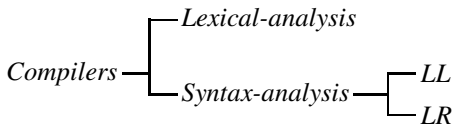
In other words, the $\leq SC$-ordering is the inverse of set-inclusion and the most abstract software description simply is the empty set of keywords:

$$\top = \emptyset$$

For this approach to be sufficiently discriminant, $Keyword$ must be more elaborate than a mere enumeration of keywords. Normally, classification power is increased by structuring the keyword domain on the basis of a hierarchical *taxonomy* of subjects, as specified *e.g.* as follows

$$Tax \cong Subject \hookrightarrow Tax \tag{5}$$

in the SETS notation. For instance, taxonomy



is described by the following recursive object of Tax:

$$\left(\begin{array}{c} \textit{Compilers} \\ \left(\begin{array}{cc} \textit{Lexical-analysis} & \textit{Syntax-analysis} \\ (\ ) & \left(\begin{array}{cc} LL & LR \\ (\ ) & (\ ) \end{array} \right) \end{array} \right) \end{array} \right)$$

Assuming a given taxonomy of subjects, SCs descriptions become more informative by specializing atomic keywords into non-empty lists of subjects [3]:

$$Keyword = Subject^{+}$$

This elaboration of the keyword-based approach is the basis of the well-known Dewey *decimal classification system* in which subjects are abbreviated to numbers and subject items are separated by dots, *e.g.* $508.03.12$ instead of $<508, 03, 12>$. For description economy, SC descriptor sets should be filtered with respect to keyword prefixes. For instance, in SC description

$$x = \{508.03.02, 548.13.92, 508.03\}$$

keyword 508.03 is redundantly recorded because it is implicit in (=it is a prefix of) another keyword, namely $508.03.02$.

[3] Of course, such lists of subjects are bound to be proper paths of the underlying taxonomy, as is the case of *e.g.* <*LR*, *Syntax-analysis*, *Compilers*> in the example above, but not the case of *e.g.* <*LR*, *Compilers*>.

In this version of the keyword-based system, the basis of SC comparison is the computation of the *longest common prefix* between two keywords,

$$lcp : Keyword \times Keyword \longrightarrow Keyword$$

upon which we have

$$comp(x, y) = filter(\{lcp(k_1, k_2) |\ k_1 \in x \wedge k_2 \in y\})$$

where $filter$ denotes the keyword *prefix filtering* function which prevents keyword redundancy [4].

2.2 The Attributive Approach

In this approach, SCs are modelled by attribute tuples, *i.e.* mappings from a domain of *attributes* to a domain of *values*:

$$SC = Attributes \hookrightarrow Values \tag{6}$$

This description technique encompasses not only record-based descriptors (*e.g.* as in the conventional relational database model) but also *objects* framed into a class-based inheritance system, in which $Values$ can also include procedure values (vulg. methods) — a currently very popular perspective.

In this case, in order to obtain a "measure" of what two SCs x and y have in common one should select every common attribute of both x and y bearing the same value:

$$comp(x, y) = \left(\begin{array}{c} a \\ x(a) \end{array} \right)_{a \in dom(x) \cap dom(y) \wedge x(a) = y(a)}$$

Clearly, the most abstract SC is the empty tuple, that is

$$\top = (\quad)$$

and the SC $\leq$-ordering is the inverse of the mapping definedness ordering:

$$x \leq y \stackrel{def}{=} dom(y) \subseteq dom(x) \wedge \forall a \in dom(y) : x(a) = y(a)$$

However, the "strictness" of this approach to SC-comparison is well-known (recall that it has been in use for a long time in *e.g.* relational databases), entailing a too "flat" *specialize/generalize* hierarchy. This problem has been tentatively overcome by allowing attribute comparison operators more liberal than strict equality ($=$), *e.g.* textual pattern matching operators.

A hybrid between this approach and the keyword-based approach is obtained by making

$$\begin{array}{rcl} Values & = & Keyword \\ & = & Subject^{+} \end{array}$$

above, within the context of a given taxonomy (or possibly one taxonomy per attribute). For instance, the SC descriptor of an LL-table-driven parser written for the Microsoft ver:7.00 C++ compiler might map attribute $code$ to keyword $C{+}{+}.Microsoft.700$ and map attribute $subject$ to another keyword, *e.g.* $parser.LL.tableDriven$.

In this approach we have

$$comp(x, y) = \left(\begin{array}{c} a \\ lcp(x(a), y(a)) \end{array} \right)_{a \in dom(x) \cap dom(y) \wedge lcp(x(a), y(a)) \neq <>}$$

[4] Because it is not the purpose of this report to be thoroughly concerned with correctness issues, namely with proving properties (1) to (4) for each classification schema, the formal definition of lcp and $filter$ are not relevant here. They are given in appendix A as a mere illustration of SETS's functional notation. The interested reader may check that, in fact, filtering keyword prefixes is a pre-requisite for correctness — check *e.g.* axiom (1).

and $\top$ is as above. This is known as the *enumerative* classification scheme [16] and it combines the advantages of both the *keyword as list of subjects* approach and the attributive approach [5].

2.3 The Faceted Perspective

As Prieto Dĭaz and Freeman [16] explain, classifying by attribute tuples as in the *enumerative* classification scheme is not problem free. Subject *synonyms* can produce different descriptors for the same component. To avoid duplicate or ambiguous descriptors one needs to have some kind of *vocabulary control* mechanism. A *term-thesaurus* is then needed to group all synonyms under similar concepts. However, subject similarity is "fuzzy" in the sense that it cannot simply be decided in the boolean domain of "yes" (full similarity) or "no" (no similarity at all). Similarity should perhaps be "measured" in the range of 0% to 100%. For instance, a vocabulary control expert may decide that subject add is 70% similar to sum (arithmetic operator) and only 30% similar to $insert$ (*e.g.* insertion of records in a database). Vocabulary control can thus be achieved on the basis of the following data structure,

$$ConceptGraph = (Subject \times Subject) \hookrightarrow 100$$

— formally a "fuzzy" binary relation on $Subject$ — leading to a so-called *multifaceted* classification schema [16, 17] based on fuzzy concept-network reasoning. In this approach, a software component is characterized, as in the attribute-based approach, by a tuple of entries, one per *facet* identifier:

$$SC = Facet \hookrightarrow Keyword$$

We simply have

$$Keyword = Subject$$

but an underlying $ConceptGraph$ is assumed enabling a function

$$fcomp : Subject \times Subject \times ConceptGraph \longrightarrow Subject$$

for computing the "closest" (conceptual) subject which compares two given subjects. Naturally, the $comp$ function for SC-comparison will resort to $fcomp$ and need to be parameterized by an underlying $ConceptGraph$, say g,

$$comp_g(x, y) = \left(\begin{array}{c} a \\ fcomp(x(a), y(a), g) \end{array} \right)_{a \in dom(x) \cap dom(y)}$$

This classification schema extends conventional enumerative classification since conceptual graphs extend subject taxonomies (which are acyclic, universally upper-bounded, left-linear graphs) with fuzzy reasoning, but within obvious limits [6].

3 An Alternative Formal Comparison Strategy

In the sections above, some standard (library) classification strategies were revised. We now move towards an alternative (and hopefully better) strategy, specially tuned to software component classification. Our starting point is the fact that the keyword-based, attributive, enumeration or faceted

[5] Formally, it can be stated that the *enumerative* classification scheme extends the *keyword as list of subjects* approach, because, by taking the range $rng(x)$ of a given descriptor $x \in SC$ (6), one obtains a set of *keywords as lists of subjects*. Note that the keywords in $rng(x)$ are not necessarily filtered by prefix, a fact which proves useful in common classification practice.

[6] For $fcomp(x, y, g)$ to be univocally determined, if two or more concepts generalize both x and y at a minimal conceptual distance, such ambiguity must be resolved at higher conceptual distances. Graph g is thus bound to be acyclic and universally upper-bounded by one unique, conceptually farthest subject, say $anything$, of which every other subject is a synonym.

classification schemes all share a common property: they are based on an "exogenous" characterization of the software artifact. By an *exogenous characterization* we mean a "black-box" description, that is, a description which does not say "what" the object "is" but rather how it can be viewed from the outside.

Better software classification schemes are obtained if the description method is *endogenous*, that is, if it grasps the semantics of the component "from the inside". Because of its complexity, size or redundancy, it is unthinkable to grasp such semantics directly at source code level. Some kind of *reverse* mechanism is required mapping each piece of code into a notation framework in which the $comp$ function can be written mathematically. In the sequel, we describe the use of formal semantics for this purpose, expressed in the SETS notation and supported by its associated calculus.

3.1 What "is" a Software Component?

In order to achieve the main purpose of this report — an illustration of the use of formal calculi in software component knowledge elicitation and classification — the basic idea will be to start from an "easy" formal notion of a *software component* and show how to build up a *repository* of such components, re-usable in a formal approach to compositional software specification. In some respects, the approach will resemble Schött's [19]. But we start from a deliberate analogy with standard hardware design strategies which has proved useful elsewhere [10], where compositional design is supported by a *graphical notation*, similar to hardware diagrams, for capturing what has become known as *horizontal composition* [6] in the software literature.

In short, a *software component* is identified with its formal *model*, *i.e.* a local *state* equipped with *operations* which access and modify the state's current value. State classes are described by (possibly recursive) set-theoretical expressions — such as *e.g.* $DecTree$ (56) or $GenDia$ (57) in appendix A — and may be subject to *invariants*, *i.e.* properties of states which should be preserved by the operations. Operations are denoted by two-state predicates (*i.e.* "input/output" relations, vulg. *pre/post conditions*) which may resort to auxiliary functions or predicates and may involve arguments and results.

This standard notion of a formal model is extended with a so-called *interface* which exhibits the model's operationality. Interfaces play two rôles in the approach. Informally, an interface resembles an "integrated circuit diagram" whose "pins" (*cf.* ports or buttons) correspond to operations (actions). For instance, a "graphical" analogy between the $STACK$ software interface and the $IC\ 74163$ hardware interface can be made apparent by comparing figures 1 and 2 [7].

Formally, interfaces behave like *types* in the adopted functional approach to horizontal composition of models, which is taken from Burstall and Lampson's work on PEBBLE [4]. That is to say, a component (formal model) is always a *value* of some *type* (interface), and the composition of n components of types I_1 to I_n, in order to build a "larger" component of type I, is expressed by a function

$$f : I_1 \times \ldots \times I_n \longrightarrow I$$

An introductory example is run in [10] which composes two $STACKS$ to specify a $FOLDER$, a component whose interface is described by figure 3 and exhibits the following functionality:

$INIT$ — initialization;

$IN@SH$ — sheet insertion;

$SH@FD$ — sheet forward;

$SH@BK$ — sheet backwards;

$\overline{READ}$ — show current sheet;

[7] Operations which produce output are "negated" (*e.g.* $\overline{POP}$) as in the CCS notation [8], paving the way to a behavioural view of components. In fact, in [10] every interface is equiped with a *behaviour* expression involving the component's operations. Here, behaviours are omitted because of the technical elaboration required by behavioural composition.

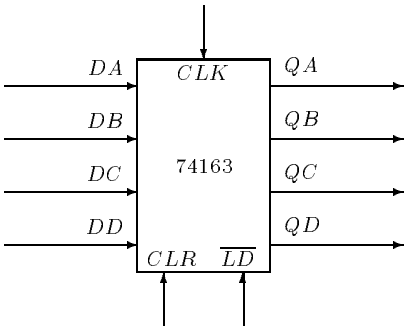


Figure 1: A hardware component interface

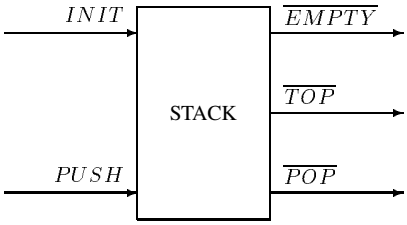


Figure 2: $STACK$ — a software component interface

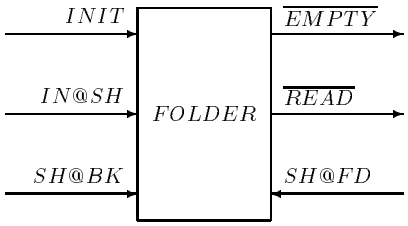


Figure 3: $FOLDER$ — another software component interface

$\overline{EMPTY}$ — check for empty folder.

The "graphical specification" of a value of "type" $FOLDER$ is described in figure 4. The corresponding compositional operator

$$f : STACK \times STACK \longrightarrow FOLDER \tag{7}$$

can be inferred from this figure by resorting to the formal meaning of the graphical notation (see [10] for details).

3.2 The Classify-by-Data Perspective

3.2.1 Syntatical Approach

The structure of the internal state of a given SC, say X, will be denoted by Σ_X or simply abbreviated by X alone, should no confusion arise from the abbreviation. Σ_X is always a SETS expression involving data constructors such as (see appendix A) $\times$, $+$, $\hookrightarrow$, $*$ *etc.*, constants such as 0, 1, 2, $\mathbb{N}$, *etc.*, as well as formal variables $A, B, C, \ldots$. For instance, concerning the internal state of component $STACK$ (figure 2), the specifier might have written

$$\Sigma_{STACK} = A^*$$

where variable A denotes the abstract type of elements to be stacked in each instantiation of $STACK$, for instance,

$$\Sigma_{STACKofNAT} = \mathbb{N}^*$$

Clearly, any functionality ("pin") we define for $STACK$ is automatically defined for $STACKofNAT$ [8]. In fact, the latter can always exhibit *more* functionality than the former, because there is additional information in its state description — if about abstract domain A we know nothing, about its instantiation $\mathbb{N}$ we know a lot of operators and properties. For instance, we could endow $STACKofNAT$ with another "pin", say $INCRTOP$, for incrementing the top of the stack, say x, *i.e.* for replacing x by $x + 1$.

[8] We may say that $STACKofNAT$ "inherits" the functionality of $STACK$. Such an analogy with object-oriented programming will, however, not be pursued further.

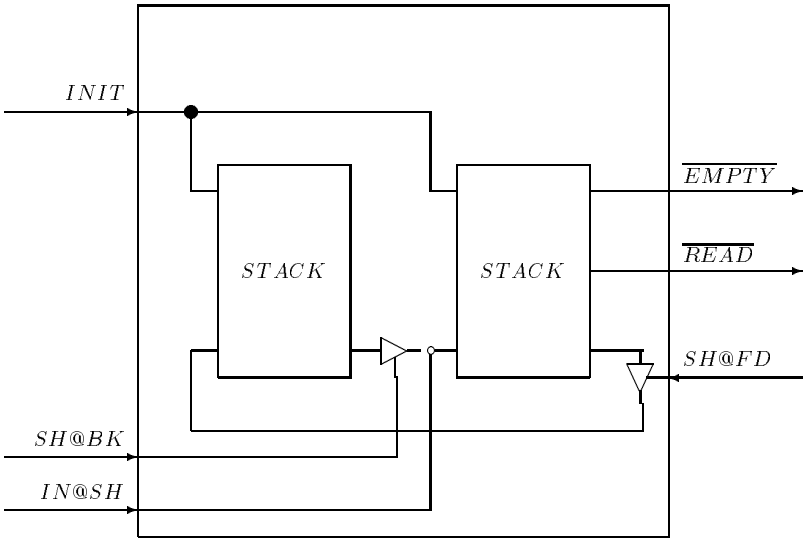


Figure 4: Compositional specification of $FOLDER$ in terms of two $STACK$s

In summary, $STACKofNAT$ *specializes* $STACK$ in the sense of section 2, that is,

$$STACKofNAT \leq STACK$$

Formally, given two components X and Y, we have

$$Y \leq X$$

(read every "Y *is a* X") if and only if

$$\Sigma_Y = \rho(\Sigma_X) \tag{8}$$

where ρ is a *substitution* of variable symbols by SETS subexpressions. For instance, given

$$\Sigma_Y = (A \times \mathbb{N}) \times B + A$$

and

$$\Sigma_X = A \times B + C$$

we have $Y \leq X$ under substitution

$$\rho = \begin{pmatrix} A & B & C \\ A \times \mathbb{N} & B & A \end{pmatrix}$$

In summary, SC specialization can be formally expressed by conventional instantiation of parameterized state descriptions in SETS notation, as one might expect.

In this framework, the comparison $comp(X, Y)$ of two given SCs X and Y simply is the *unification* of formal expressions Σ_X and Σ_Y [9]. $\top$, the most abstract component, is such that $\Sigma_\top = A$, where A is any SETS variable. $\top$ is the SC about which very few actions can be specified — its specifies just a (parameterized) "memory cell" where one can but *write* or *read* [10].

[9] That is, the "greatest SETS expression" from which both Σ_X and Σ_Y can be derived by substitutions, *cf.* ρ_1 and ρ_2 (respectively) in the diagram of figure 5. By "greatest" one means that any other generalization Z common to both X and Y will also be a generalization of $comp(X, Y)$. In categorial terms one might say that $comp(X, Y)$ is the *product* of X and Y.

[10] Cf. component RSC in [10].

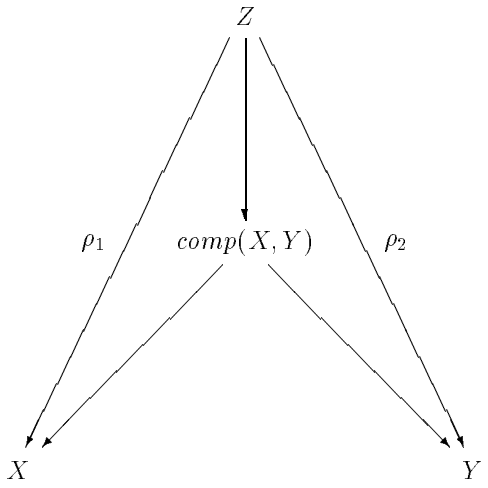


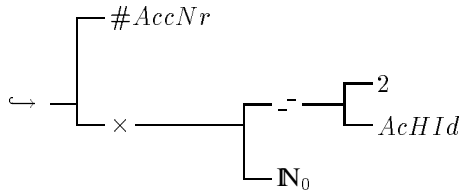
Figure 5: Diagram for $comp(X, Y)$

Example

Consider the following pair of "toy" components: $BAMS$ (*bank account management system*) and $LCAT$ (*library catalogue*). The former is specified as follows:

$$\begin{array}{lll} \Sigma_{BAMS} = \#AccNr \hookrightarrow & 2^{AcHId} & \times \quad \textit{/*set of account holders */} \\ & \mathbb{N}_0 & \quad\;\; \textit{/*account balance */} \end{array}$$

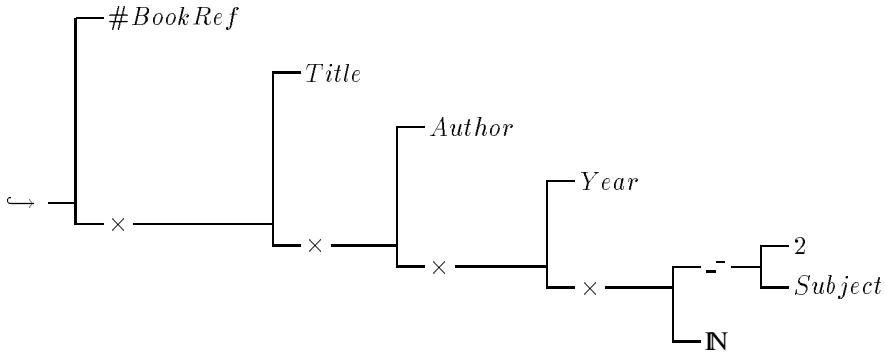
that is, Σ_{BAMS} is SETS expression:



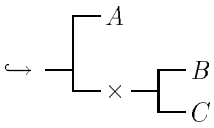
The latter is specified by

$$\begin{array}{lll} \Sigma_{LCAT} = \#BookRef \hookrightarrow & Title & \times \quad \textit{/*book title */} \\ & Author & \times \quad \textit{/*book author */} \\ & Year & \times \quad \textit{/*book date */} \\ & 2^{Subject} & \times \quad \textit{/*classification */} \\ & \mathbb{N} & \quad\;\; \textit{/*book ISBN */} \end{array} \tag{9}$$

that is, its internal state class is SETS expression:



Their comparison may be expressed by the SETS expression which follows,



that is,

$$\Sigma_{comp(BAMS,LCAT)} = A \hookrightarrow B \times C \tag{10}$$

for substitutions

$$\rho_{BAMS} = \begin{pmatrix} A & B & C \\ \#AccNr & 2^{AcHId} & \mathbb{N}_0 \end{pmatrix}$$

$$\rho_{LCAT} = \begin{pmatrix} A & B & C \\ \#BookRef & Title & Author \times Year \times 2^{Subject} \times \mathbb{N} \end{pmatrix}$$

Of course, the choice of variables A, B and C is arbitrary. Writing $X \hookrightarrow Y \times Z$ would have the same effect as writing $A \hookrightarrow B \times C$. So, strictly speaking, in this approach $\leq$ is a pre-order and not a partial-order. Equivalence among SCs, obtained by $\leq$ anti-simmetric closure, means "the same up to capture free variable substitution".

3.2.2 Semantic Approach

However interesting, the specialization ordering and comparison function of the previous section are too strict and not sufficiently powerful. Many components are distinguished which should be found equivalent. In fact, we don't want to distinguish between two components whose data are "equivalent", *e.g.* in C,

```
struct X {                          struct X {
    A first;                            struct Y {
    struct Y {                              A first;
        B first;       equivalent to        B second;          (11)
        C second;                       } first;
    } second;                           C second;
};                                  };
```

If we resort to the analogy of table 1 in order to check what the SETS counterpart of (11) is, we obtain

$$A \times (B \times C) \text{ equivalent to } (A \times B) \times C$$

But this is just associative law (58) of the SETS calculus (*cf.* appendix B) where $\cong$ denotes the desired equivalence relation on SETS expressions [11]. Therefore, two seemingly different components may actually be "the same" component up to some valid transformations. In the example above, and because $\times$ is not only associative but also a commutative SETS construct, *cf.* law (59) we may subject part of (9) to the following transformations,

$$\begin{aligned}(Title \times Author \times Year) \times 2^{Subject} \times \mathbb{N} &= \\ 2^{Subject} \times (Title \times Author \times Year) \times \mathbb{N} &= \\ 2^{Subject} \times (Title \times Author \times Year \times \mathbb{N})\end{aligned}$$

obtaining

$$\Sigma_{LCAT} = \#BookRef \hookrightarrow 2^{Subject} \times (Title \times Author \times Year \times \mathbb{N})$$

By unifying this version of Σ_{LCAT} with Σ_{BAMS} we obtain an intersection "larger" than (10),

$$\Sigma_{comp(BAMS,LCAT)} = A \hookrightarrow 2^B \times C$$

that is, which captures more of their similarity. A more interesting example follows.

Another Example

Consider, from appendix A, specifications $DecTree$ (56) and $GenDia$ (57) which are apparently incomparable data models specified within unrelated application domains. As a matter of fact,

$$comp(DecTree, GenDia) = \top$$

if criterion (8) is adopted. Now, consider the following reasoning in the SETS calculus: firstly, we re-write (56) avoiding semantically biased symbols such as $DecTree$, $What$, $Answer$ and using more "neutral" variable symbols such as T, I, A:

$$T \cong I \times (A \hookrightarrow T) \tag{12}$$

Now let $A = 2$. Then

$$\begin{aligned}I \times (A \hookrightarrow T) &\cong I \times (2 \hookrightarrow T) \\ &\cong I \times (T+1)^2 \\ &\cong I \times (T+1) \times (T+1)\end{aligned}$$

cf. laws (64) and (61). So, (12) becomes

$$T \cong I \times (T+1) \times (T+1) \tag{13}$$

Let us now compare (13) with (57). Clearly, (13) and (57) are the "same" data model up to variable renaming. We conclude that *genealogical diagrams* "are" specializations of *decision trees*,

$$GenDia \leq DecTree$$

that is, if a package implementing the latter is available, then it may be *re-used* to implement the former.

[11] For simplicity symbol "=" may occasionally be used instead of "$\cong$".

Of course, a full comparison between both models should examine the algorithmic structure as well — as is discussed in section 6 [12] — but the point of this example is just to show the usefulness of SETS in comparing or assessing data model specification. The important point is that this leads to a broader specialization (pre) ordering among components which takes the calculus into account:

$$Y \leq X \quad \text{iff} \quad \exists \rho : \Sigma_Y \cong \rho(\Sigma_X)$$

and, in a formal sense, defines a basic *similarity* relation between software components able to support their analogical reuse across unrelated domain areas.

Further Examples

Many examples of the usefulness of $\cong$-based comparison could be given, providing evidence of considerable overlapping among software data models. For instance, if one instantiates $I = 1$ in specification (12) of an abstract decision tree, and since 1 is the identity of $\times$ — *cf.* law (60) — we obtain

$$\begin{aligned} T &\cong 1 \times (A \hookrightarrow T) \\ &\cong A \hookrightarrow T \end{aligned}$$

which is but the Tax (5) model for taxonomies, specified in section 2.1 of this report. This means that *taxonomies* are another specialization of decision trees. In fact, they can be regarded as decision trees with a NIL question at each node.

Still more suggestive is perhaps the fact that Tax also specializes a general purpose data model, the one specifying hierarchical *file systems* (such as UNIX or MS/DOS):

$$\begin{aligned} FS &\cong Id \hookrightarrow (File + Dir) && \text{/*FS = file system */} \\ & && \text{/*Id = identifier */} \\ & && \text{/*File = file contents */} \\ Dir &\cong FS && \text{/*Dir = directory,} \text{ i.e. } \textit{sub-file-system} \text{ */} \end{aligned}$$

In short, we write

$$FS \cong Id \hookrightarrow (File + FS)$$

By making $File = 0$ and because 0 is the identity of $+$, see law (62), we obtain

$$\begin{aligned} FS &\cong Id \hookrightarrow (0 + FS) \\ &\quad Id \hookrightarrow FS \end{aligned}$$

which is (up to variable renaming) the same data model as Tax (5), confirming our intuition: if we build a file system with no files at all, such a nested structure of directories is but a taxonomy of file-system identifiers.

Figure 6 depicts the hierarchy of SC's which encompasses the examples given so far (using uppercase letter identifiers to denote the corresponding SC's names, *e.g.* $DECTREE$ for $DecTree$ *etc.*). It should be noted that other intermediate components may exist which are not explicitly recorded in the diagram, which is a typical "is-a" hierarchy with multiple inheritance. For instance, any operation specified for FS or $DECTREE$ can be — suitably transformed via the relevant $\cong$-morphism, which may be regarded as a mere "format converter" — inherited by TAX.

[12] For instance, making decisions in $DecTree$ corresponds exactly to ascending genealogical data in $GenDia$, for a fixed *menu* (set of available "answers") at every "decision" level: $2 \cong \{father, mother\}$.

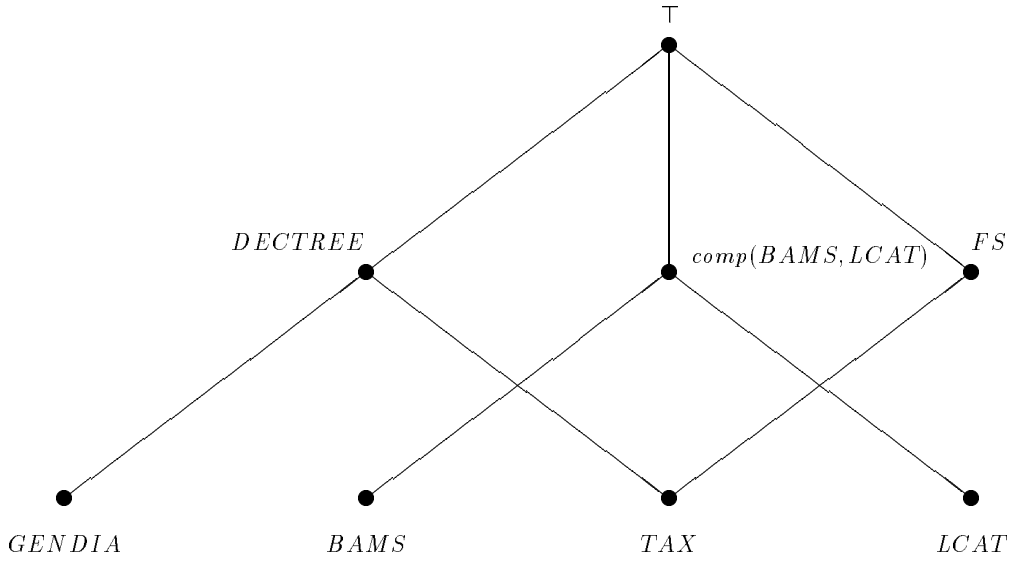


Figure 6: Example SC's hierarchy

3.3 Additional Structural Relationships among Software Components

The support of a formal notation such as SETS makes it possible to define further relationships (typical of software repository organization) other than "is a", at the same level of rigour.

- **Decomposition** ("A is used by B" or "A is part of B") — this relationship holds between two components A and B wherever the internal state of B is a tuple, a component of which can be individualized as being of the same class as A's internal state:

$$\Sigma_B \cong \ldots \times \Sigma_A \times \ldots$$

- **Particularization** ("A is a particular case of B" or "A is an exceptional specialization of B") — this relationship holds between two components A and B wherever

$$\Sigma_A \cong (\Sigma_B)_\phi$$

  where ϕ is a SETS predicate (invariant). Because of this invariant's preservation not every operation available at B-level can be inherited by A; in a sense, A "is an exceptional" B.

- **Implementation** or refinement ("A implements B") — this relationship holds between two components A and B wherever every element of Σ_B can be represented by at least one item of Σ_A, possibly involving a representation invariant (ϕ) on Σ_A. In SETS this is expressed by writing

$$\Sigma_B \trianglelefteq (\Sigma_A)_\phi$$

  meaning "Σ_B is reified by Σ_A". Since the $\trianglelefteq$-algebra is fairly well developed in the SETS literature, a wide range of this kind of relationship among components can be established in a formal way.

3.3.1 Desired Properties

Many properties of the above relationships can be postulated, and proved or refuted in SETS, *e.g.*transitive properties:

$$\frac{C \text{ is used by } B,\ B \text{ is used by } A}{C \text{ is used by } A}$$

$$\frac{C \text{ is particular case of } B,\ B \text{ is particular case of } A}{C \text{ is particular case of } A}$$

compatibility properties:

$$\frac{X \text{ is used by } A,\ Y \text{ implements } X}{(A \text{ where } Y \text{ replaces } X) \text{ implements } A}$$

non-monotonic inheritance:

$$\frac{B \text{ is particular case of } A}{CONTEXT(B) \text{ is } \textbf{not} \text{ a } CONTEXT(A)}$$

etc. A thorough study of the approach (naturally omitted here for the sake of brevity) should of course develop such a "repository calculus" as much as possible.

4 Application to ERA-diagram Classification

A major challenge for the strategy described so far for software artifact classification is how to classify the very many pieces of software which have come to existence in the past with no dedicated formal specification support. This problem is in fact enormous, should we consider the wide variety of design methodologies which have been adopted for software design in the last decades.

Clearly, one should avoid proliferation of *calculi* for different design notations (*e.g.* Structured Programming, ERA-diagrams, *etc.*). The strategy in this report will be to adopt the SETS calculus as common denominator, that is, to

- map each notation onto SETS abstract notation (note that in some cases this will resemble a "reverse engineering" process);
- reason in SETS;
- map results back to original notation.

The remainder of this report focuses on the application of this strategy to reasoning and classifying relational database schemata described by hierarchical ERA ("Entity-Relationship-Attribute") diagrams. Clearly, the major task to be carried out is to map ERA-diagrams into the SETS notation, in a structural way. Such a task can be regarded as defining a *formal semantics* for such diagrams.

4.1 First Step: A SETS Semantics for ERA-diagrams

This section presents the strategy to reverse ERA-diagrams into the SETS notation introduced in [13] and later developed in [18] and [5]. It is a formalization of the usual interpretation given to ERA diagrams in Relational Database Design.

Following [3], the following steps should be followed in the logical design of a relational database:

Step 1. Each Simple Entity is translated to a table. Simple entities are entities which are not subtypes nor have their own subtypes.

Step 2. Each attribute is translated to a column of the same name.

- Optional attributes are translated to *null columns* (columns whose fields may take the 'null' value).
- Mandatory attributes are translated to *non null columns*.

Step 3. The components of the entity's unique identifier become the primary key of the corresponding table. Note that each entity may be uniquely identified by more than one identifier — the one which is most used is normally chosen.

Step 4. Many to one (and one to one) relationships (N:1) become foreign keys. That is, they become a copy of the unique identifier of each entity that is referred to from the 'one' end.

Optional relationships lead to *null columns*.

Mandatory relationships lead to *non null columns*.

Step 5. Index Design:

Create indices for:

- the Primary Key (unique index);
- each of the Foreign Keys;
- those suggested by any Function/Attribute matrix.

Step 6. Design for Sub-types:

An entity sub-type is simply an entity with its own attributes or relationships, which also inherits every attribute or relationship from its parent entity (super-type), and so on upwards the super-type hierarchy. There are two basic alternatives for the design of sub-types, each one having its own advantages and disadvantages:

- All in one table;
- One table for each sub-type.

All in one table A table is built for the outer entity and, optionally, relational views [13] may be created for each of its sub-types. As before, attributes and 'N:1' relationships create columns for data and foreign keys.

The above process is repeated for each sub-type and each of its sub-types, and so on. The difference is every column created for a sub-type is made *null* (optional). A mandatory attribute or relationship for a sub-type would not be applicable for another.

One table for each sub-type Tables are built for each sub-type in order to cover all possible instances. In case of several levels of sub-typing, it is normal, in the first level, to create one table for each sub-type. At the downer levels, views could then be created.

Step 7. Exclusive Relationships:

This case will not be addressed here.

□

A collection of generic entities A, B, ..., A_i, ...will be considered in the sequel such that generic entity A has primary key $\#A$ and attributes a_1, a_2, ...a_n. Every entity A_i has primary key $\#A$ and attributes a_{i1}, a_{i2} ...a_{in}. B is considered to have primary key $\#B$ and attributes b_1, b_2, ...b_k. Finally, a generic relationship R will be considered, having attributes r_1, r_2, ...r_p.

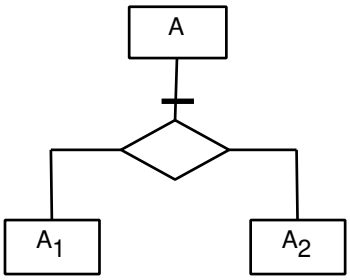


Figure 7: Sub-type Entities.

4.1.1 Sub-types

Consider, in figure 7, a diagram depicting an entity (A) with two sub-types (A_1 and A_2). A possible set of relational tables implementing this diagram, following the steps above, is:

$$\begin{aligned} Table\ A: &\quad (\underline{\#A}, a_1, a_2, \ldots, a_n) \\ Table\ A_1: &\quad (\underline{\#A}, a_{11}, a_{12}, \ldots, a_{1k}) \\ Table\ A_2: &\quad (\underline{\#A}, a_{21}, a_{22}, \ldots, a_{2m}) \end{aligned}$$

which is bound to satisfy the following integrity constraint:

> The sets of key values present respectively in tables A_1 and A_2 have to be disjoint and both have to be a subset of the set of key values of table A. (14)

that is, tables A_1 and A_2 record only the attributes specific to the corresponding subtypes.

As is well-known, 'tables' in the relational model implement 'relations' in the mathematical sense. Therefore, one may use law (66) in reversing the above tables to the SETS notation and obtaining sets of tuples (relations) each of which bearing an invariant predicate — fdp (74) — indicating that there exists a *functional dependency* between the key of each table and the other attributes of the same table:

$$\begin{aligned} A &\cong 2^{\#A \times (a_1 \times a_2 \times \ldots \times a_n)}_{fdp} \\ A_1 &\cong 2^{\#A \times (a_{11} \times a_{12} \times \ldots \times a_{1k})}_{fdp} \\ A_2 &\cong 2^{\#A \times (a_{21} \times a_{22} \times \ldots \times a_{2m})}_{fdp} \end{aligned}$$

A database consisting of the above three tables may be reversed to a data sort in SETS which is the product of the three preceding relations, and globally satisfies the invariant inv-$subtype'$ which mirrors what has been stated in (14):

$$\begin{aligned} \Sigma' &\cong (A \times A_1 \times A_2)\text{inv-}subtype' \\ &\cong (2^{\#A \times (a_1 \times a_2 \times \ldots \times a_n)} \\ &\quad \times 2^{\#A \times (a_{11} \times a_{12} \times \ldots \times a_{1k})} \\ &\quad \times 2^{\#A \times (a_{21} \times a_{22} \times \ldots \times a_{2m})})\text{inv-}subtype' \end{aligned}$$

where

$$\begin{aligned} \text{inv-}subtype'(r_1, r_2, r_3) \stackrel{def}{=} \quad & fdp(r_1) \wedge fdp(r_2) \wedge fdp(r_3) \wedge \\ & (\pi_1[r_2] \cup (\pi_1[r_3]) \subseteq \pi_1[r_1] \wedge \\ & \pi_1[r_2] \cap \pi_1[r_3] = \emptyset \end{aligned}$$

[13] A 'view' is a mode of accessing part of a table as if it were another table.

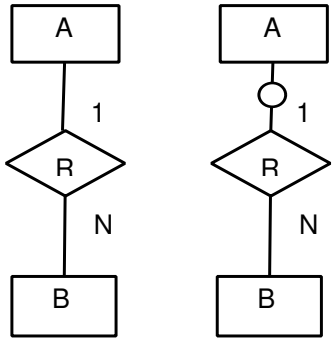


Figure 8: '1:N' Relationships, mandatory at the 'N' end.

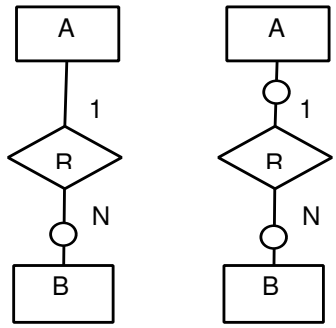


Figure 9: '1:N' Relationships, optional at the 'N' end.

By reducing Σ' through law (66) one obtains:

$$\begin{aligned}\Sigma \quad \cong \quad & ((\#A \hookrightarrow a_1 \times a_2 \times \ldots \times a_n) \\ & \times (\#A \hookrightarrow a_{11} \times a_{12} \times \ldots \times a_{1k}) \\ & \times (\#A \hookrightarrow a_{21} \times a_{22} \times \ldots \times a_{2m}))\text{inv-}subtype \end{aligned} \tag{15}$$

where

$$\begin{aligned}\text{inv-}subtype(f_1, f_2, f_3) \quad \stackrel{def}{=} \quad & dom(f_2) \subseteq dom(f_1) \\ & \wedge\, dom(f_3) \subseteq dom(f_1) \\ & \wedge\, dom(f_2) \cap dom(f_3) = \emptyset \end{aligned} \tag{16}$$

In summary, the SETS semantics of the diagram of figure 7 is the SETS structure (15) subject to invariant (16).

4.1.2 '1:N' Relationships

Figures 8 and 9 depict 4 diagrams, each of which illustrating two entities A and B related by a '1:N' relationship (R). The four diagrams differ only in the optionality of the relationship between the entities. The first diagram of figure 8 imposes each instance of entity A to correspond to N instances of entity B, with $N \neq 0$, while the second diagram only expresses that instances of A may be related to N instances

of B, $N \neq 0$. In other words, the first diagram forces each instance of entity B to correspond to an instance of entity A, altogether exhausting all instances of A. By contrast, the second diagram does not impose the participation of all instances of A in the relationship.

The first diagram of figure 9 means that each instance of A is related to N instances of B ($N \geq 0$) and, the second, that each instance of A may be related to N instances of B ($N \geq 0$). In other words, diagrams of figure 9 tell that each instance of B may be related to N instances of A, the first enforcing the participation of all instances of A in the relationship, while the second does not enforce such a constraint [14].

By applying the steps described above to any of the diagrams of figures 8 and 9, the following tables are obtained:

$$Table\ A: \quad (\underline{\#A}, a_1, a_2, \ldots, a_n) \tag{17}$$

$$Table\ B: \quad (\underline{\#B}, b_1, b_2, \ldots, b_k, \#A) \tag{18}$$

The integrity constraint to be imposed on these tables depends on the diagram which is being taken into consideration, as follows.

The first diagram of figure 8 imposes:

> The set of values in table B which refer to rows of table A (column $\#A$), must be the same as the set of key values of table A. Moreover, the values of the column $\#A$ in table B must be 'non null', that is, all the rows of table B must refer to a row of table A. (19)

The second diagram of figure 8 imposes:

> The set of values in table B which refer to rows of table A (column $\#A$), must be a subset of the set of key values of table A. Moreover, the values of the column $\#A$ in table B must be 'non null', that is, all the rows in table B must refer to a row of table A. (20)

The first diagram of figure 9 imposes:

> The set of values in table B which refer to rows of table A (column $\#A$) must be the same as the set of key values of table A. Moreover, the values of the column $\#A$ in table B may be 'null', that is, there may exist rows in table B which do not refer to any row of table A. (21)

The second diagram of figure 9 imposes:

> The set of values in table B which refer to rows of table A (column $\#A$) must be a subset of the set of key values of table A. Moreover, the values of the column $\#A$ of table B may be 'null', that is, there may exist rows in table B which do not refer to any row of table A. (22)

The database consisting of tables (17) and (18) above is thus equivalent to:

$$Table\ A: \quad (\underline{\#A}, a_1, a_2, \ldots, a_n) \tag{23}$$

$$Table\ B: \quad (\underline{\#B}, b_1, b_2, \ldots, b_k) \tag{24}$$

$$Table\ R: \quad (\underline{\#B}, \#A) \tag{25}$$

subject to the following integrity constraint:

[14] The differences between these diagrams will be better understood in the sequel, when attaching characteristic invariants to each diagram.

- first diagram of figure 8:

  The set of values in table R which refer to rows of table A (column $\#A$), must be the same as the set of key values of table A. The set of values in table R which refer to rows of table B (column $\#B$), must be the same as the set of key values of table B. (26)

- second diagram of figure 8:

  The set of values in table R which refer to rows of table A (column $\#A$), must be a subset of the set of key values of table A. The set of values in table R which refer to rows of table B (column $\#B$), must be the same as the set of key values of table B. (27)

- first diagram of figure 9:

  The set of values in table R which refer to rows of table A (column $\#A$), must be the same as the set of key values of table A. The set of values in table R which refer to rows of table B (column $\#B$), must be a subset of the set of key values of table B. (28)

- second diagram of figure 9:

  The set of values in table R which refer to rows of table A (column $\#A$), must be a subset of the set of key values of table A. The set of values in table R which refer to rows of table B (column $\#B$), must be a subset of the set of key values of table B. (29)

Now, by converting to the SETS notation the database formed by tables (23), (24) and (25) one obtains:

$$\begin{aligned}\Sigma' &\cong (A \times B \times R)\text{inv-}rel1N' \\ &\cong (2^{\#A \times (a_1 \times a_2 \times \ldots \times a_n)} \\ &\quad \times 2^{\#B \times (b_1 \times b_2 \times \ldots \times b_k)} \\ &\quad \times 2^{\#B \times \#A})\text{inv-}rel1N'\end{aligned}$$

where inv-$rel1N'$ will depend again on the integrity constraints imposed by each diagram.

By reducing Σ' through law (66) (for retrieve function $mkf \times mkf \times mkf$), one obtains:

$$\begin{aligned}\Sigma \cong\ & ((\#A \hookrightarrow a_1 \times a_2 \times \ldots \times a_n) \\ & \times (\#B \hookrightarrow b_1 \times b_2 \times \ldots \times b_k) \\ & \times (\#B \hookrightarrow \#A))\text{inv-}rel1N\end{aligned} \tag{30}$$

subject to the invariant (case by case):

- First diagram of figure 8:

$$\begin{aligned}\text{inv-}rel1N(f_1, f_2, f_3) \stackrel{def}{=}\ & rng(f_3) = dom(f_1) \\ & \wedge\ dom(f_3) = dom(f_2)\end{aligned} \tag{31}$$

- Second diagram of figure 8:

$$\begin{aligned}\text{inv-}rel1N(f_1, f_2, f_3) \stackrel{def}{=}\ & rng(f_3) \subseteq dom(f_1) \\ & \wedge\ dom(f_3) = dom(f_2)\end{aligned} \tag{32}$$

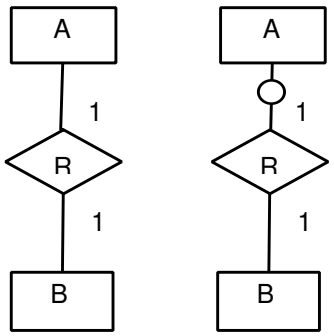


Figure 10: '1:1' Relationships.

- First diagram of figure 9:

$$\text{inv-}rel1N(f_1, f_2, f_3) \stackrel{def}{=} rng(f_3) = dom(f_1) \wedge dom(f_3) \subseteq dom(f_2) \tag{33}$$

- Second diagram of figure 9:

$$\text{inv-}rel1N(f_1, f_2, f_3) \stackrel{def}{=} rng(f_3) \subseteq dom(f_1) \wedge dom(f_3) \subseteq dom(f_2) \tag{34}$$

4.1.3 '1:1' Relationships

Relationships 1:1 between two entities are a particular case of 1:N relations. As can be seen in figure 10, each instance of entity A corresponds to no more than one instance of entity B. That is, one has the following tables:

$$\begin{aligned} Table\ A: &\quad (\underline{\#A}, a_1, a_2, \ldots, a_n) \\ Table\ B: &\quad (\underline{\#B}, b_1, b_2, \ldots, b_k, \#A) \end{aligned}$$

under the following integrity constraint:

> The set of values in table B which refer to rows of table A (column $\#A$), must be a subset of the set of key values of table A. Moreover, the values of the column $\#A$ of table B must be 'non null', *i.e.*, all the rows of table B must refer to a row of the table A, that is the set of values of column $\#A$ of table B must be the same as the set of key values of table A. (35)

The database consisting of the two tables above is equivalent to:

$$Table\ A: \quad (\underline{\#A}, a_1, a_2, \ldots, a_n) \tag{36}$$

$$Table\ B: \quad (\underline{\#B}, b_1, b_2, \ldots, b_k) \tag{37}$$

$$Table\ R: \quad (\underline{\#B}, \#A) \tag{38}$$

subject to integrity constraint:

> The set of values of table R which refer to rows of table A (column $\#A$ of table R), must be a subset of the set of key values of table A. The set of values in table R which refer to rows of table B (column $\#B$), must be the same as the set of key values of table B. (39)

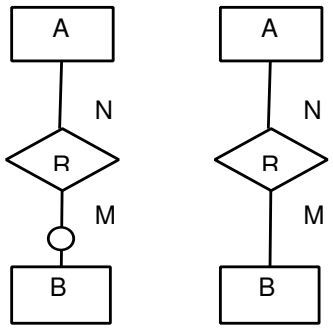


Figure 11: 'N:M' relationships, mandatory at the 'N' end.

By reversing tables (36), (37) and (38) to the SETS notation, it comes:

$$\begin{aligned}\Sigma' &\cong (A \times B \times R)\text{inv-}rel11' \\ &\cong (2^{\#A\times(a_1\times a_2\times\ldots\times a_n)}_{fdp} \\ &\quad \times 2^{\#B\times(b_1\times b_2\times\ldots\times b_k)}_{fdp} \\ &\quad \times 2^{\#B\times\#A}_{fdp})\text{inv-}rel11'\end{aligned}$$

Once again, by reducing Σ' through law (66) and retrieve function $(mkf \times mkf \times mkf)$, one obtains:

$$\begin{aligned}\Sigma &\cong ((\#A \hookrightarrow a_1 \times a_2 \times \ldots \times a_n) \\ &\quad \times (\#B \hookrightarrow b_1 \times b_2 \times \ldots \times b_k) \\ &\quad \times (\#B \hookrightarrow \#A))\text{inv-}rel11\end{aligned}$$

where

$$\begin{aligned}\text{inv-}rel11(f_1, f_2, f_3) \stackrel{def}{=}\ & rng(f_3) \subseteq dom(f_1) \\ & \wedge\, dom(f_3) = dom(f_2)\end{aligned}$$

4.1.4 'N:M' Relationships

Figures 11 and 12 depict four diagrams representing, each one, two entities related to each other by 'many to many' relationship ($N : M$). The four diagrams differ in the optionality of the relationship. So, in the first diagram of figure 11 and in the first diagram of figure 12 all instances of one of the entities must be related with some instances of the other. The other entity may have instances which are not related to anything. In the second diagram of figure 11, all instances of both entities must be related with instances of the other one. Moreover, in the second diagram of figure 12, entities A and B may have instances related to zero or more instances of the other one.

Following [3], 'N:M' relationships are dispensable as they can be expressed by two '1:N' relationships and a secondary entity. Figure 13 is a suitable illustration: the attributes of the secondary entity in *Invoice_Entry* are the same as the ones in the relationship R, and relationships $R1$ and $R2$ have no attributes.

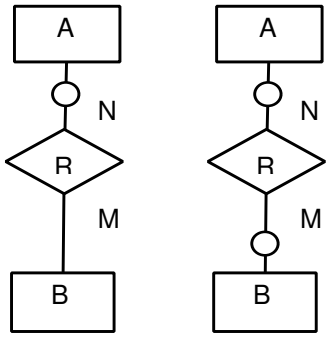


Figure 12: ‘N:M’ relationships, optional at the ‘N’ end.

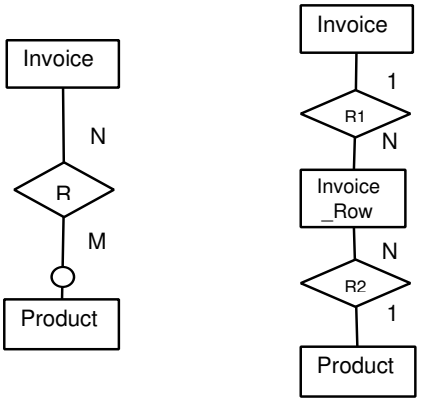


Figure 13: Two equivalent diagrams, illustrating the dispensability of the ‘N:M’ relationships.

Nevertheless, relational designs may involve 'N:M' relationships. By applying the strategy described in [1] to any of the diagrams of figures 11 and 12, the following tables are obtained:

$$Table\ A: \quad (\underline{\#A}, a_1, a_2, \ldots, a_n) \tag{40}$$

$$Table\ B: \quad (\underline{\#B}, b_1, b_2, \ldots, b_k) \tag{41}$$

$$Table\ R: \quad (\underline{\#B, \#A}) \tag{42}$$

whose integrity constraints vary for each of the diagrams being considered:

- The first diagram of figure 11 imposes:

  > The set of values of column $\#A$ of table R must be the same as the set of key values of table A. Moreover, the set of the values in column B of table R must be a subset of the set of the key values of table B. (43)

- The second diagram of figure 11 imposes:

  > The set of values of column $\#A$ of table R must be the same as the set of key values of table A. Moreover, the set of values of column B in table R must be the same as the set of key values of table B. (44)

- The first diagram of figure 12 imposes[15]:

  > The set of values of column A in table R must be a subset of the set of key values of table A. Moreover, the set of the values of column B in table R must be the same as the set of key values of table B. (45)

- The second diagram of figure 12 imposes:

  > The set of values of column $\#A$ in table R must be a subset of the set of key values of the table A. Moreover, the set of the values of column B in the table R must be a subset of the set of key values of table B. (46)

By reversing the database formed by tables (40), (41) and (42) to the SETS notation, one obtains:

$$\begin{array}{rcl} \Sigma' & \cong & (A \times B \times R)\text{inv-}relNM' \\ & \cong & (2^{\#A \times (a_1 \times a_2 \times \ldots \times a_n)} \\ & & \times\ 2^{\#B \times (b_1 \times b_2 \times \ldots \times b_k)} \\ & & \times\ 2^{\#A \times \#B})\text{inv-}relNM' \end{array}$$

Now reducing Σ' through law (66) — for retrieve function $mkf \times mkf \times id$ — one obtains:

$$\begin{array}{rcl} \Sigma & \cong & ((\#A \hookrightarrow a_1 \times a_2 \times \ldots \times a_n) \\ & & \times\ (\#B \hookrightarrow b_1 \times b_2 \times \ldots \times b_k) \\ & & \times\ 2^{\#A \times \#B})\text{inv-}relNM \end{array}$$

where, for each case, one has:

- first diagram of figure 11:

$$\text{inv-}relNM(f_1, f_2, \rho) \stackrel{def}{=} \pi_1[\rho] = dom(f_1) \wedge \pi_2[\rho] \subseteq dom(f_2)$$

[15] The integrity constraint imposed by the first diagram of figure 12 is identical to the one imposed by the first diagram of figure 11, exchanging the roles of entities A and B.

- second diagram of figure 11:

$$\text{inv-}relNM(f_1, f_2, \rho) \stackrel{def}{=} \pi_1[\rho] = dom(f_1) \wedge \pi_2[\rho] = dom(f_2)$$

- first diagram of figure 12:

$$\text{inv-}relNM(f_1, f_2, \rho) \stackrel{def}{=} \pi_1[\rho] \subseteq dom(f_1) \wedge \pi_2[\rho] = dom(f_2)$$

- second diagram of figure 12:

$$\text{inv-}relNM(f_1, f_2, \rho) \stackrel{def}{=} \pi_1[\rho] \subseteq dom(f_1) \wedge \pi_2[\rho] \subseteq dom(f_2)$$

4.2 Second Step: Structure Abstraction in SETS

After obtaining from a given ERA-diagram a first structure in SETS, it is necessary to abstract it (generalize it), in order to factor out accidental detail and emphasize structural relations between the entities [16] Such an abstraction (or generalization) process on SETS data structures proceeds as follows:

1st step Collapse all sub-expressions only involving primitive data sorts (typically, attribute domains) in independent sort variables [17]. For example, in the following structure in SETS

$$\Sigma \cong (\#Number \hookrightarrow Name \times Address \times Telephone) \times (\#Car \hookrightarrow 2^{\#Number \times Hours \times Kms})$$

sub-expression $Name \times Address \times Telephone$ can be collapsed into a single variable A and sub-expression $Hours \times Kms$ can be collapsed into another variable B:

$$\Sigma' \cong (\#Number \hookrightarrow A) \times (\#Car \hookrightarrow 2^{\#Number \times B}) \tag{47}$$

2nd step Replace key identifiers by variables, in order to obtain a structure without direct references to the real system and, thus, more abstract. In the example above, from (47) one might obtain *e.g.*:

$$\Sigma'' \cong (\#N \hookrightarrow A) \times (\#C \hookrightarrow 2^{\#N \times B})$$

3rd step Apply to the structure obtained in the previous step as many SETS laws as possible, from right to left, in particular laws capable of lifting invariants from the structure.

That is, starting from an abstract structure E_ϕ, by applying this step one may obtain a structure E' such that

$$E' \trianglelefteq E_\phi$$

For example, taking the structure Σ obtained in (30) subject to invariant (31) or (32), it is possible to reverse Σ through law (67), involving retrieve function $(id \times \bowtie)$, to obtain:

$$\Sigma_1 \cong ((\#A \hookrightarrow a_1 \times a_2 \times \ldots \times a_n) \times (\#B \hookrightarrow b_1 \times b_2 \times \ldots \times b_k \times \#A))_{\text{inv-}R1N_1} \tag{48}$$

where,

[16] Recall that ERA-entities always originate, when reversed to SETS, "outermost" finite mappings.
[17] That is to say, never collapse entity keys.

- in case of (31), one obtains:

$$\text{inv-}R1N_1(f_1, f_2) \stackrel{def}{=} \pi_{k+1}[rng(f_2)] = dom(f_1) \tag{49}$$

- in case of (32), one obtains:

$$\text{inv-}R1N_1(f_1, f_2) \stackrel{def}{=} \pi_{k+1}[rng(f_2)] \subseteq dom(f_1)$$

such that

$$\Sigma_1 \trianglelefteq \Sigma$$

The role of the invariant to lift is, of course, determinant. For instance, should (30) be considered afected by invariant (33) or (34), than the obtained generalization would have been, now applying law (69),

$$\begin{aligned} \Sigma_2 \quad \cong \quad & ((\#A \hookrightarrow a_1 \times a_2 \times \ldots \times a_n) \\ & \times (\#B \hookrightarrow b_1 \times b_2 \times \ldots \times b_k \times (\#A + 1)))_{\text{inv-}R1N_2} \end{aligned} \tag{50}$$

where, using the corresponding retrieve function $id\times \bowtie$ *cf.* (72),

- in case of (33) one obtains:

$$\text{inv-}R1N_2(f_1, f_2) \stackrel{def}{=} \neg is_Nil(\pi_{k+1}(rng(f_2))) \Rightarrow \pi_{k+1}(rng(f_2)) = dom(f_1) \tag{51}$$

- in case of (34) one obtains:

$$\text{inv-}R1N_2(f_1, f_2) \stackrel{def}{=} \neg is_Nil(\pi_{k+1}(rng(f_2))) \Rightarrow \pi_{k+1}(rng(f_2)) \subseteq dom(f_1) \tag{52}$$

such that

$$\Sigma_2 \trianglelefteq \Sigma$$

Appendix C contains a quick reference list of rules for converting ERA-diagram to the SETS notation which take this process of invariant lifting into account.

5 An Example

Consider figure 14 which depicts an ERA-diagram for a "toy factory" production database. From this diagram one can infer the following design intuitions:

- the factory builds some kind of *equipment* whose production involves individual *components* obtained from a second source;
- each individual component has a *cost* and is stocked in some *quantity* controlled with respect to an arbitrated minimum *alarm* level;
- of every component is, in a fixed quantity, *part of* at least one *equipment* (*e.g.* PC board ref. `X` has n integrated circuits of ref. `Y`);
- equipments may contain, in a fixed quantity, other equipments as *sub-blocks* (*e.g.* personal computer ref. `Z` has m PC boards of ref. `X`).

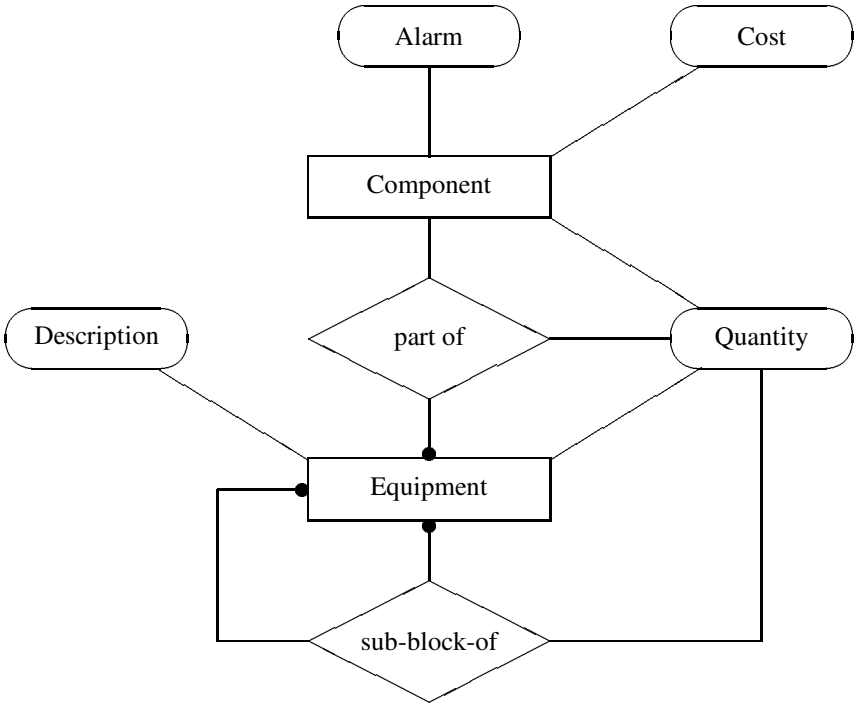
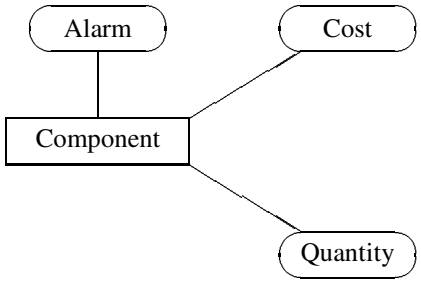


Figure 14: An ERA-diagram for a “Toy Factory” Production Database

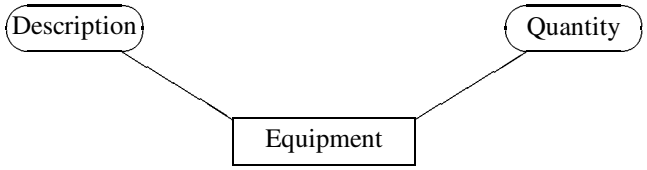
In summary, a production tree must exist for each equipment which may involve individual components and/or other production sub-trees.

The SETS counterpart of this diagram can be obtained via the rules developed above and compiled in appendix C. There are only two entities in the diagram, $Component$ and $Equipment$. Using rule (78) one obtains:



$$= \quad \#Component \hookrightarrow Alarm \times Cost \times Quantity$$

and



$$= \quad \#Equipment \hookrightarrow Description \times Quantity$$

The *part of* relationship can be taken into account by adding, via rule (88), an extra finite mapping:

$$\begin{array}{l}(\#Component \times \#Equipment \hookrightarrow Quantity) \times \\ (\#Component \hookrightarrow Alarm \times Cost \times Quantity) \times \\ (\#Equipment \hookrightarrow Description \times Quantity)\end{array}$$

Finally, relationship *sub-block-of* is incorporated using rule (90):

$$\begin{array}{ll}(& (\#Equipment \times \#Equipment \hookrightarrow Quantity) \times \\ & (\#Component \times \#Equipment \hookrightarrow Quantity) \times \\ & (\#Component \hookrightarrow Alarm \times Cost \times Quantity) \times \\ & (\#Equipment \hookrightarrow Description \times Quantity) \\ )_{\Phi} & \end{array} \tag{53}$$

subject to an overall invariant Φ induced by the rules used above:

$$\begin{array}{rcl}\Phi(r_1, r_2, e_1, e_2) & \stackrel{def}{=} & \pi_1[dom(r_1)] \subseteq dom(e_1) \\ & & \wedge \\ & & \pi_2[dom(r_1)] \subseteq dom(e_1) \\ & & \wedge \\ & & \pi_1[dom(r_2)] = dom(e_2) \\ & & \wedge \\ & & \pi_2[dom(r_2)] \subseteq dom(e_1)\end{array}$$

Independently of this invariant, by laws (65), (63) and (59) expression (53) can be transformed into

$$\begin{array}{l}((\#Equipment \times (\#Equipment + \#Component)) \hookrightarrow Quantity) \times \\ (\#Equipment \hookrightarrow Description \times Quantity) \times \\ (\#Component \hookrightarrow Alarm \times Cost \times Quantity)\end{array}$$

The same invariant can be partly lifted via law (68) finally leading to:

$$\begin{array}{lll}(& (\#Equipment \hookrightarrow & Description \times Quantity \times \\ & & ((\#Equipment + \#Component) \hookrightarrow Quantity) \\ & (\#Component \hookrightarrow & Alarm \times Cost \times Quantity) \\ )_{\varphi} & & \end{array} \quad (54)$$

where the invariant remaining is

$$\begin{array}{lll}\varphi(e_1, e_2) & \stackrel{def}{=} & let \quad X = \bigcup_{k \in dom(e_1)} dom(\pi_3(e_1(k))) \\ & & \quad\;\; E = \{x \in \#Equipment | \; \langle 1, x\rangle \in X\} \\ & & \quad\;\; C = \{x \in \#Component | \; \langle 2, x\rangle \in X\} \\ & & in \quad dom(e_2) = C \wedge E \subseteq dom(e_1)\end{array} \quad (55)$$

It should be noted that what the ERA-diagram meaning amounts to — denoted in SETS by (54) and (55) — is, on formal specification grounds, a poor specification of what the specifier might have had in mind but could not express in the diagram. For instance, where is in (55) a clause enforcing that *no equipment can recursively be a sub-block of itself*[18]? It is interesting to consider (54) in the context of the law for recursion removal presented in [11], for $\#Equipment$ the relevant key domain, whereby one may "reverse" the first mapping in (54) by introducing recursion:

$$X \cong Description \times Quantity \times ((X + \#Component) \hookrightarrow Quantity)$$

But this step can only be accomplished if the above clause for acyclic equipment nesting holds. So, in summary, the starting ERA-diagram is able to record the $\#Equipment$-key-definedness constraint but fails to record so essential a property as acyclicness (otherwise we would get infinite production trees).

For classification purposes, it is now understood that the "core" of our "toy factory" production database can be abstracted by the following SETS expression,

$$X \cong A \times ((X + C) \hookrightarrow B)$$

which would clearly deserve, in the repository of figure 6, the status of a "new" component.

6 Conclusions and Further Work

The main target of this report is to provide evidence that formal program calculi such as SETS [9, 11] can be useful not only to develop software but also

- to extract "knowledge" from existing software artifacts;
- to reason about such formal knowledge descriptions, *e.g.* by comparing them, calculating their "difference", *etc.*;
- to effectively classify software components according to an underlying formal classification strategy, as opposed to conventional 'ad hoc' approaches.

[18] In [10] it is shown how a simple algebra of multi-sets (bags) provides for a compact and elegant specification of interesting operations of this problem such as *e.g. part explosion*.

The first point above can either be regarded as a formal semantics exercise or a reverse engineering process, depending on the context.

The strategy has been applied by Cruz [5] to about twenty real-size industrial information systems whose designs could be found in non-confidential workstudy-in-industry reports available from the library of University of Minho. Only data-dictionaries and ERA-diagrams were reversed, in the style of the exercise of section 5 above. The task of "rationalizing" the abstract description of such a collection of software systems into a reuse-effective SETS-hierarchy is still in progress. But is already clear that such systems repeat each other in some degree. This is natural, after all, since they were developed by independent companies. However, and as everybody seems to fear, such a redundancy tends to crop up inside the same software repository.

The fact that company models partly repeat each other seems to stem from the standardization of management technology itself. What appears to be wrong is the "fine grain" style of reusability that seems to be common practice, normally confined to loose relational schemata correlated informally. Is the approach put forward in this report the beginning of what might be called, using industrial terminology, an *algebra of company models*? We hope at least to be able to identify a few widely reusable "coarse grain" *company model parts*.

The approach has the merit of discussing and equating similarity among software artifacts across the boundaries normally imposed by conventional *domain area* oriented classification. A few paths for future work are left open by this report (let alone some technically ill-supported claims which need to be formally proved or refuted.) Perhaps the most relevant has to do with the adopted "classify-by-data" criterion. Although it may seem that the most discriminant part of a SC is its internal state structure, there is a need for a still larger equivalence among SCs taking the operations in account and leading to a coarser "classify-by-observational-equivalence" criterion, a topic which is currently being investigated elsewhere [14, 15].

References

[1] Allen C. P. *Effective Structured Techniques — From Strategy to CASE.* Prentice Hall, New York, 1991.

[2] Asymetrix Corporation *Using Open Script - An Introduction and Reference to the Open Script Language.* 1989.

[3] Barker R. *CASE*METHOD — Entity Relationship Modelling.* Addison-Wesley Publishing Company, Great Britain, 1992.

[4] Burstall R.M., Lampson B. *A Kernel Language for Modules and Abstract Data Types.* Technical Report, Digital Systems Research Center, Palo Alto, September 1984.

[5] Cruz A.M. *Engenharia Reversa em Larga Escala de Sistemas de Informa ão.* Workstudy Report at INESC Group 2361, December 1993.

[6] Goguen J.A. *Reusing and Interconnecting Software Components.* IEEE Computer, 19(2), 16–28, February 1986.

[7] Jones C.B. *Systematic Software Development Using* VDM. Prentice-Hall International, Series in Computer Science (C.A.R.Hoare Ed.), 1986.

[8] Milner R. *Communication and Concurrency.* Prentice-Hall International, Series in Computer Science (C.A.R.Hoare Ed.), 1989.

[9] Oliveira J.N. *A Reification Calculus for Model-Oriented Software Specification.* Formal Aspects of Computing, Vol.2, 1-23, 1990, Springer-Verlag.

[10] Oliveira J.N. *Especificação Formal de Programas.* Lecture Notes for M.Sc. Course in Computing, University of Minho, 3^{rd} edition, 1993.

[11] Oliveira J.N. *Software Refinement by Calculation in* SETS. Invited communication to BCS FACS 5th Refinement Workshop, *Theory and Practice of Formal Software Development*, 8–10 January 1992, London, 140-171, Springer Verlag.

[12] Oliveira J.N. *Desenvolvimento Formal de 'Software'*, Lecture Notes for M.Sc. Course in Computing, University of Minho, 1993.

[13] Oliveira J.N. *Components & Compositionality in SOUR.* SOUR Meeting, INESC Lisbon, 25 February 1993.

[14] Oliveira R.C. *Assessment and Reorganization of Object-Oriented Software Class Systems — A Case Study.* Technical Report WP-C19-2A, INESC Group 2361, Braga, December 1993.

[15] Oliveira R.C. *Subtipos e Subclasses no Paradigma de Orientação por Objectos.* M.Sc. Thesis (in preparation), University of Minho, Braga, Portugal.

[16] Prieto-Diáz R., Freeman P. *Classifying Software for Reusability.* IEEE Software, 4-1, January 1987.

[17] Prieto-Diáz R. *Implementing Faceted Classification for Software Reuse.* Comm. of the ACM, 34(5):89–97, May 1991.

[18] Rodrigues C. *Sobre o Desenvolvimento Formal de Bases de Dados.* M.Sc. Thesis, University of Minho, Braga, Portugal, 1993.

[19] Schött O. *A Theory of Program Modules, their Specification and Implementation (Extended Abstract).* Internal Report CSR-155-83, University of Edinburgh, December 1982.

[20] Spanoudakis G., Constantopoulos P. *Similarity for Analogical Software Reuse: A Conceptual Modelling Approach.* NATURE Report Series 92-08, ESPRIT Project 6353.

[21] Spivey J.M. *The Z Notation — A Reference Manual.* Prentice-Hall International, Series in Computer Science (C.A.R.Hoare Ed.), 1989.

A An Overview of the SETS Notation

A limited number of set-theoretical constructs is enough for modelling in SETS fairly elaborate abstract objects. Such "primitive" constructs are,

- *cartesian product* of two sets A and B (*cf.* records):

$$A \times B = \{\langle a, b\rangle |\ a \in A \wedge b \in B\}$$

- *disjoint union* of two sets A and B (*cf.* variant records):

$$A + B = (\{1\} \times A) \cup (\{2\} \times B)$$

- *exponentiation* (A raised to a finite set B, see examples below):

$$A^B = \{f |\ f : B \rightarrow A\}$$

An analogy between these basic constructs and C language data structuring constructs is given in table 1.

On top of these one can build the following *derived* constructs:

Table 1: Analogy between C and SETS

SETS	C	**Description**

$X = A \times B$	`struct X {`
	`  A first;`
	`  B second;`
	`};`
		'Records'

$X = A + B$	`union X {`
	`  int tag; /* 1,2 */`
	`  A ifA;`
	`  B ifB;`
	`};`
		Variant 'Records'

$X = B^A$	`B[A]`	'Arrays'
$A + 1$	`A *...`	'Pointers'

- finite subsets of a finite set A:

$$2^A$$

- finite binary relations from A to B:

$$2^{A \times B}$$

- finite mappings from A to B:

$$A \hookrightarrow B = \bigcup_{K \subseteq A} B^K$$

- finite sequences on a set A:

$$A^\star = \bigcup_{n \geq 0} A^{\overline{n}}$$

  where exponent $\overline{n}$ denotes the initial segment of $\mathbb{N}$ whose cardinality is n, that is, the set $\{1, 2, \ldots, n\}$. For simplicity, we will write n instead of $\overline{n}$, and therefore 0 instead of the empty set $\emptyset$. We also write A^+ as an abbreviation of $A^* - \{<>\}$, that is, to denote non-empty finite sequences on a set A.

- union types:

$$A + 1$$

  where 1 stands for any singleton set, *cf.* [9, 11]. Typically, $1 \cong \{NIL\}$.

A last ("meta") construct is recursive definition,

$$X \cong \mathcal{F}(X)$$

where X is the name of a data sort and $\mathcal{F}$ is a SETS-expression ("functor") involving the above primitive or derived constructs.

Recursive definitions are a powerful mechanism for specifying arbitrarily elaborate data structures. For example, the following C syntax for simple *decision trees*,

```
typedef struct DecTree{
        char *What; /* Question or Decision */
        subTrees *R;
        } ;
```

```
typedef struct subTrees {
        node *first;
        struct subTrees *next;
        } ;

typedef struct node {
        char *Answer;
        decTree *SubTree;
        } ;
```

(where decisions are modelled by `DecTree` nodes with no answers) is specified in the SETS notation simply as follows:

$$DecTree \cong What \times (Answer \hookrightarrow DecTree) \tag{56}$$

Another example mentioned in this report is the following C syntax fragment for *genealogical diagrams* (the "pedigree view" of family relationship):

```
typedef struct GenDia {
        Ind individual;         /* data about an individual */
        struct GenDia *father;  /* genealogy of his/her father
                                   (if known) */
        struct GenDia *mother;  /* genealogy of his/her mother
                                   (if known) */
        } ;
```

which is easily "reversed" into SETS notation as follows, *cf.* table 1:

$$GenDia \cong Ind \times (GenDia + 1) \times (GenDia + 1) \tag{57}$$

In the opposite direction, Table 1 can also be used for coding purposes. Consider, for instance, the following SETS specification of a linked list of objects A,

$$\begin{aligned} L &\cong 1 + Node \\ Node &\cong A \times L \end{aligned}$$

that is, a "linear equation"

$$L \cong 1 + A \times L$$

From table 1 one draws that, in C, every $x \in L$ will be declared as

```
Node *x;
```

since $L = 1 + Node$. Thus a datatype definition is required for `Node`, which is immediate from table 1 again:

```
typedef struct Node {
        A first;
        struct Node *next;
        } ;
```

SETS adopts a notation for recursive functions which is standard in mathematics, *e.g.*

$$x! = \begin{cases} x = 0 & \Rightarrow \quad 1 \\ \neg(x = 0) & \Rightarrow \quad x \times (x - 1)! \end{cases}$$

instead of

$$\begin{array}{rl} x! = & if\ \ x = 0 \\ & then\ \ 1 \\ & else\ \ x \times (x-1)! \end{array}$$

The basic notation includes all set-theoretical operators typical of constructive specification languages (such as *e.g.* VDM [7] or Z [21]) dealing with lists (A^*), sets (2^A), mappings ($A \hookrightarrow B$) and so on. The following function specifying the *longest common prefix* between two lists (used in section 2.1) illustrates the list-algebra:

$$lcp(k_1,k_2) \stackrel{def}{=} \left\{ \begin{array}{rcll} (k_1 = <> \vee k_2 = <>) & \Rightarrow & <> & \\ \neg((k_1 = <> \vee k_2 = <>)) & \Rightarrow & let & h_1 = head(k_1) \\ & & & h_2 = head(k_2) \\ & & & t_1 = tail(k_1) \\ & & & t_2 = tail(k_1) \\ & & in & \left\{ \begin{array}{rcl} h_1 = h_2 & \Rightarrow & cons(h_1, lcp(t_1,t_2)) \\ \neg(h_1 = h_2) & \Rightarrow & <> \end{array} \right. \end{array} \right.$$

The following function, also used in section 2.1, illustrates the set-algebra:

$$filter(S) \stackrel{def}{=} \left\{ \begin{array}{rcll} S = \{\} & \Rightarrow & \{\} & \\ \neg(S = \{\}) & \Rightarrow & let & e \in S \\ & & in & \left\{ \begin{array}{rcl} (\exists x \in S : x < e) & \Rightarrow & filter(S - \{e\}) \\ \neg((\exists x \in S : x < e)) & \Rightarrow & \{e\} \cup filter(S - \{e\}) \end{array} \right. \end{array} \right.$$

Many illustrations of the mapping algebra occur in the main text of this report.

B A Summary of the SETS Calculus

This appendix is not a comprehensive listing of all laws of the SETS calculus — it includes only the laws referred to in the main text. For further information about SETS the reader is referred to [9, 11, 12]

B.1 Equational Calculus Fragment

$$A \times (B \times C) \cong (A \times B) \times C \tag{58}$$
$$A \times B \cong B \times A \tag{59}$$
$$A \times 1 \cong A \tag{60}$$
$$\underbrace{A \times \ldots \times A}_{n} \cong A^n \tag{61}$$
$$A + 0 \cong A \tag{62}$$
$$A \times (B + C) \cong (A \times B) + (A \times C) \tag{63}$$
$$A \hookrightarrow B \cong (B + 1)^A \tag{64}$$
$$(B + C) \hookrightarrow A \cong (B \hookrightarrow A) \times (C \hookrightarrow A) \tag{65}$$

B.2 Inequational Calculus Fragment

$$A \hookrightarrow B \quad \trianglelefteq^{fdp}_{mkf} \quad 2^{A \times B} \tag{66}$$
$$A \hookrightarrow (B \times C) \quad \trianglelefteq^{eqd}_{\bowtie} \quad (A \hookrightarrow B) \times (A \hookrightarrow C) \tag{67}$$

$$A \hookrightarrow D \times (B \hookrightarrow C) \quad \trianglelefteq^{prd}_{sjoin} \quad (A \hookrightarrow D) \times ((A \times B) \hookrightarrow C) \tag{68}$$

$$A \hookrightarrow (B \times (C+1)) \quad \trianglelefteq^{\bowtie}_{subeqd} \quad ((A \hookrightarrow B) \times (A \hookrightarrow C)) \tag{69}$$

Note that law (69) is a particular case of law (68), for $B = 1$.

B.3 Abstraction Functions

$$mkf(\rho) \stackrel{def}{=} \left(\begin{array}{c} a \\ the(\{b \in B |\ a \rho b\}) \end{array} \right)_{a \in \pi_1[\rho]} \tag{70}$$

$$\sigma \bowtie \tau \stackrel{def}{=} \left(\begin{array}{c} a \\ \langle \sigma(a), \tau(a) \rangle \end{array} \right)_{a \in dom(\sigma)} \tag{71}$$

$$\rho_1 \bowtie \rho_2 \stackrel{def}{=} \left(\begin{array}{c} x \\ \langle \rho_1(x), \left\{ \begin{array}{lcl} x \in dom(\rho_2) & \Rightarrow & i_1(\rho_2(x)) \\ \neg(x \in dom(\rho_2)) & \Rightarrow & i_2(NIL) \end{array} \right. \rangle \end{array} \right)_{x \in dom(\rho_1)} \tag{72}$$

$$sjoin(f,g) \stackrel{def}{=} \left(\begin{array}{c} a \\ \langle f(a), \left(\begin{array}{c} b \\ g(a,b) \end{array} \right)_{\langle a,b \rangle \in sel(a,g)} \rangle \end{array} \right)_{a \in dom(f)} \tag{73}$$

$$\begin{array}{rcl} & where & \\ sel(a,f) & \stackrel{def}{=} & \{\langle a', b \rangle \in dom(f) |\ a' = a\} \end{array}$$

B.4 Invariant Predicates

$$fdp(\rho) \stackrel{def}{=} \forall \langle a,b \rangle, \langle a',b' \rangle \in \rho : (a = a' \Rightarrow b = b') \tag{74}$$

$$eqd(\sigma,\tau) \stackrel{def}{=} dom(\sigma) = dom(\tau) \tag{75}$$

$$subeqd(\rho_1,\rho_2) \stackrel{def}{=} dom(\rho_2) \subseteq dom(\rho_1) \tag{76}$$

$$prd(f,g) \stackrel{def}{=} \pi_1[dom(g)] \subseteq dom(f) \tag{77}$$

C An ERA-diagram Semantics Cookbook

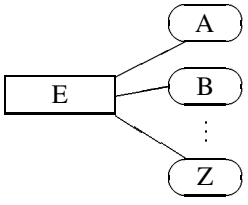
This appendix presents a quick reference list of rules for converting ERA-diagram to the SETS notation.

In the ERA-diagrams which follow, symbols E, F denote entities; symbols $\#E$, $\#F$ denote the relevant key attributes; and A, B, C, D, G, $\cdots$, Z denote attribute domains. Each rule is of the form

$$d = e$$

where d is a ERA-diagram and e is the corresponding SETS denotation.

C.1 Entities



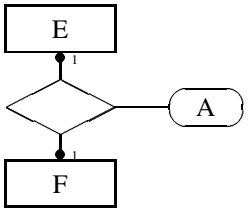
$$= \quad \#E \hookrightarrow A \times B \times \cdots \times Z \tag{78}$$

C.2 Relationships

In general we assume that an abstract attribute A is associated to each relationship, which in each case may actually be a cartesian product of concrete attributes.

1:1 Relationships Several cases to be considered depending on the optionality of the involved entities E or F:

1. E and F both optional:



$$= \quad ((\#F \hookrightarrow \#E \times A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\phi} \tag{79}$$

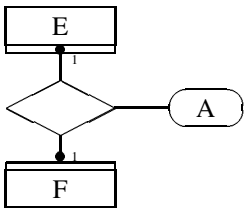
Associated invariant:

$$\phi(\rho,\sigma,\sigma') \stackrel{def}{=} \\ \forall x, x' \in dom(\rho) : x \neq x' \Rightarrow \pi_1(\rho(x)) \neq \pi_1(\rho(x'))\wedge \\ dom(\rho) \subseteq dom(\sigma')\wedge \\ \pi_1[rng(\rho)] \subseteq dom(\sigma)$$

Should there be no attribute A of the relationship ($A = 1$):

$$\#F \hookrightarrow \#E \times 1 \cong \#F \hookrightarrow \#E$$

2. E compulsory and F optional:

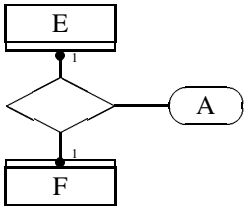


$$= \quad ((\#F \hookrightarrow \#E \times A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\phi} \tag{80}$$

Associated invariant:

$$\phi(\rho,\sigma,\sigma') \stackrel{def}{=} \\ \forall x, x' \in dom(\rho) : x \neq x' \Rightarrow \pi_1(\rho(x)) \neq \pi_1(\rho(x'))\wedge \\ dom(\rho) \subseteq dom(\sigma')\wedge \\ \pi_1[rng(\rho)] = dom(\sigma)$$

3. F compulsory and E optional:

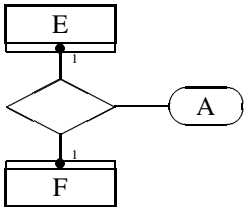


$$= \quad ((\#F \hookrightarrow \#E \times A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\phi} \tag{81}$$

Associated invariant:

$$\begin{aligned}&\phi(\rho,\sigma,\sigma') \stackrel{def}{=}\\ &\quad \forall x,x' \in dom(\rho) : x \neq x' \Rightarrow \pi_1(\rho(x)) \neq \pi_1(\rho(x'))\wedge\\ &\quad dom(\rho) = dom(\sigma')\wedge\\ &\quad \pi_1[rng(\rho)] \subseteq dom(\sigma)\end{aligned}$$

4. E and F both compulsory:



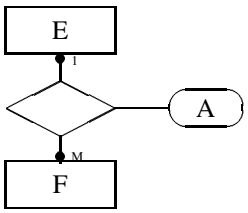
$$= \quad ((\#F \hookrightarrow \#E \times A)\times (\#E \hookrightarrow \cdots)\times (\#F \hookrightarrow \cdots))_{\phi} \tag{82}$$

Associated invariant:

$$\begin{aligned}&\phi(\rho,\sigma,\sigma') \stackrel{def}{=}\\ &\quad \forall x,x' \in dom(\rho) : x \neq x' \Rightarrow \pi_1(\rho(x)) \neq \pi_1(\rho(x'))\wedge\\ &\quad dom(\rho) = dom(\sigma')\wedge\\ &\quad \pi_1[rng(\rho)] = dom(\sigma)\end{aligned}$$

M:1 Relationships Again 4 rules, in the same order:

1.

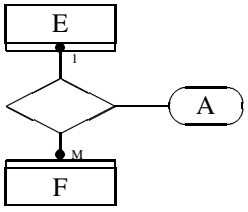


$$= \quad ((\#F \hookrightarrow \#E \times A)\times (\#E \hookrightarrow \cdots)\times (\#F \hookrightarrow \cdots))_{\gamma} \tag{83}$$

Associated invariant:

$$\begin{aligned}&\gamma(\rho,\sigma,\sigma') \stackrel{def}{=}\\ &\quad dom(\rho) \subseteq dom(\sigma')\wedge\\ &\quad \pi_1[rng(\rho)] \subseteq dom(\sigma)\end{aligned}$$

2.

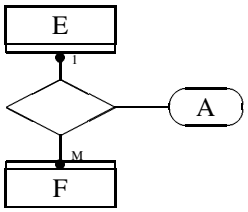


$$= \quad ((\#F \hookrightarrow \#E \times A)\times (\#E \hookrightarrow \cdots)\times (\#F \hookrightarrow \cdots))_{\gamma} \tag{84}$$

Associated invariant:

$$\begin{aligned}&\gamma(\rho,\sigma,\sigma') \stackrel{def}{=}\\ &\quad dom(\rho) \subseteq dom(\sigma')\wedge\\ &\quad \pi_1[rng(\rho)] = dom(\sigma)\end{aligned}$$

3.

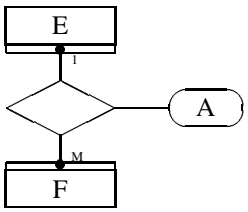


$$= \quad ((\#F \hookrightarrow \#E \times A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\gamma} \tag{85}$$

Associated invariant:

$$\gamma(\rho, \sigma, \sigma') \stackrel{def}{=} \\ dom(\rho) = dom(\sigma') \wedge \\ \pi_1[rng(\rho)] \subseteq dom(\sigma)$$

4.



$$= \quad ((\#F \hookrightarrow \#E \times A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\gamma} \tag{86}$$

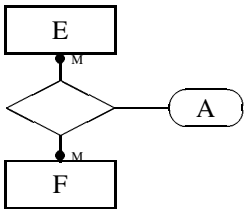
Associated invariant:

$$\gamma(\rho, \sigma, \sigma') \stackrel{def}{=} \\ dom(\rho) = dom(\sigma') \wedge \\ \pi_1[rng(\rho)] = dom(\sigma)$$

Note that $1 : 1$ relationships are particular cases of $M : 1$ relationships, only imposing an extra injectivity constraint.

M:M Relationships (in the same order):

1.



$$= \quad ((\#E \times \#F \hookrightarrow A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\varphi} \tag{87}$$

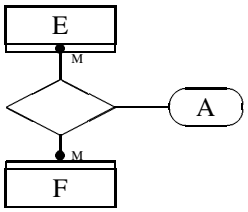
Associated invariant:

$$\varphi(\rho, \sigma, \sigma') \stackrel{def}{=} \pi_1[dom(\rho)] \subseteq dom(\sigma) \wedge \\ \pi_2[dom(\rho)] \subseteq dom(\sigma')$$

Again, for $A = 1$ (no attributes in relationship) we have

$$\#E \times \#F \hookrightarrow 1 \cong (1+1)^{\#E \times \#F} \\ \cong 2^{\#E \times \#F}$$

2.

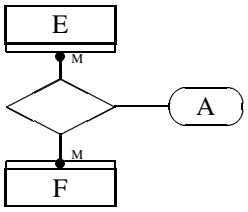


$$= \quad ((\#E \times \#F \hookrightarrow A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\varphi} \tag{88}$$

Associated invariant:

$$\varphi(\rho,\sigma,\sigma') \stackrel{def}{=} \quad \pi_1[dom(\rho)] = dom(\sigma) \wedge \\ \pi_2[dom(\rho)] \subseteq dom(\sigma')$$

3.



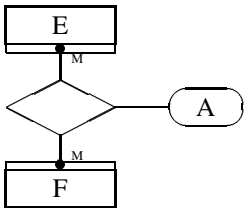
$$= \quad ((\#E \times \#F \hookrightarrow A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\varphi} \tag{89}$$

Associated invariant:

$$\varphi(\rho,\sigma,\sigma') \stackrel{def}{=} \quad \pi_1[dom(\rho)] \subseteq dom(\sigma) \wedge \\ \pi_2[dom(\rho)] = dom(\sigma')$$

(notice that this diagram is symmetric wrt. the previous diagram)

4.



$$= \quad ((\#E \times \#F \hookrightarrow A)\times \\ (\#E \hookrightarrow \cdots)\times \\ (\#F \hookrightarrow \cdots))_{\varphi} \tag{90}$$

Associated invariant:

$$\varphi(\rho,\sigma,\sigma') \stackrel{def}{=} \quad \pi_1[dom(\rho)] = dom(\sigma) \wedge \\ \pi_2[dom(\rho)] = dom(\sigma')$$

D A Tool for ERA-diagrams Conversion to SETS

This appendix presents a modest multimedia tool for assiting ERA-diagram conversion into the SETS notation. This tool has been developed in TOOLBOOK [2] for the WINDOWS environment.

The tool is, first of all, an ERA-diagram editor (*cf. e.g.* figure 18). Once a diagram has been edited, it can be converted to the SETS notation (figures 19 and 20) as well as a textual description to insert in a design document (figure 21).

Figure 16: Editing an Entity.

Figure 18: Editing an ERA Diagram.

Figure 20: Resulting SETS expression.

Figure 21: Diagram printout.