tanulmányok

150/1983

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

COMPUTER AND AUTOMATION INSTITUTE
HUNGARIAN ACADEMY OF SCIENCES

IFIP TC.2 WORKING CONFERENCE

"System Description Methodologies"

May 22-27. 1983

Kecskemét, Hungary

Tanulmányok 150/1983

A kiadásért felelős:

DR VÁMOS TIBOR

Szerkesztette:

KNUTH ELŐD

ISBN 963 311 164 1
ISSN 0324-2951

C O N T E N T

ON RELATIVE STRENGTH OF INFORMATION MODELS

Bengt Lundberg

SYSLAB

Department of Information Processing
and Computer Science

University of Stockholm
S-106 91 Stockholm, Sweden

Abstract:

The concept of information model is since long employed to denote a representation of abstract knowledge about a perceived portion of the real world. When an information model is constructed it is changed (refined) in order to be an as precise as possible description of the considered portion of the real world.

In the paper the changes applied to an information model are discussed and analyzed from a formal point of view. Thereby, it is assumed that an information model is represented in a first-order predicate logic language. Within the framework given by predicate logic the relative strength of information models, and of the constructs constituting them, are discussed and examplified. Further, it is shown that an assumption of the existence of instances of the employed predicates can be useful, and practical, in order to find incorrectnesses. It follows, in particular, that it is important to represent implicit assumptions explicitly.

This work is supported by the National Swedish Board for Technical Development (STU)

1. INTRODUCTION

The area of information modelling, or conceptual modelling, has been the objective of intense research during the last decade. A number of approaches to information modelling and, in particular, formalisms for the representation of information models have been presented, e.g. (Sen-77, Hou-79, Bub-80). During the last years first-order predicate logic has been employed as the basis for the analysis of formalisms of information models, but also as a representational formalism, e.g. (Bub-80, Rei-81, Lun-82a). In this paper we discuss and analyze from a formal point of view the process of changing an information model aiming at a more precise (or, stronger) information model with respect to an assumed universe of discourse.

The relative strength of information models, and the constructs within them, are discussed. Further, it suggests that an information model should be made as complete as possible, in particular, implict assumptions should be made explicit. Also, an existence assumption is introduced and discussed with particular attention to its application to information modelling.

2. THE CONCEPT OF INFORMATION MODEL

When a portion of the real world, usually called a universe of discourse (ISO-82), is considered, two types of knowledge about it can be identified, namely: concrete knowledge and abstract (or, general) knowledge. With concrete knowledge is meant such knowledge that refers to states-of-affairs in the universe of discourse, e.g. "Jim earns 1000". With abstract knowledge is meant such knowledge that refers to conditions holding for types of states-of-affairs, e.g. "all employees have a salary".

An information model is defined to be a representation of abstract knowledge about a universe of discourse. Thus, representations of concrete knowledge is not considered. When referring to a universe of discourse the intension is to refer to the structural properties of the perceived states of the real world, i.e. a particular set of entities is not assumed in the real world.

In what follows we will assume that an information model is represented in a first-order predicate logic language. This inplies that an information model is constituted by a number of first-order sentences which are the non-logical axioms of a first-order theory for which the universe of discourse is a set of models. Further, it is

assumed that the identity relation holds in the universe of discourse, i.e. we have theories with equality. This implies that function symbols and individual constants are dispensible, i.e. the only non-logical symbols of an information model are the predicate symbols.

3. DEVELOPMENT OF AN INFORMATION MODEL

When an information model is constructed for a universe of discourse two principal strategies can be applied:

a) define the types of state-of-affairs that are considered, i.e. define the employed predicate symbols, and then declare sentences reflecting abstract knowledge in the defined language.

b) start with a part of the universe of discourse and define the language for it and represent the abstract knowledge about it, then extend the language of the information model and the set of sentences.

This can be represented graphically as follows:

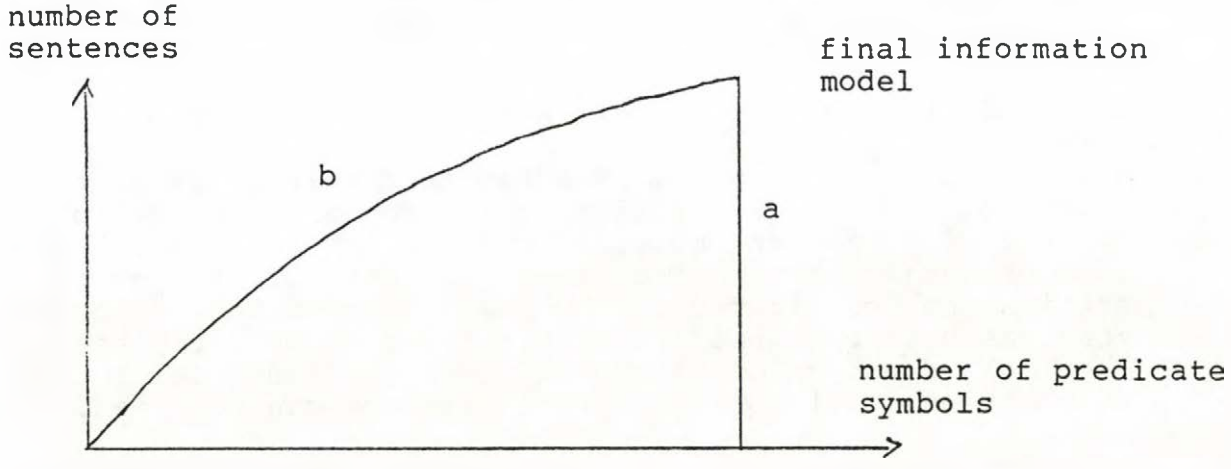


Figure 1

Strategy b is the more practical strategy, but from a theoretical point of view the interesting strategy is

strategy a. Further, strategy b can be reduced to strategy a by considering it as an interactive procedure that extends the language of the information model and adds sentences the information model. When the information models is constructed it is assumed that its sentences are true (satisfied) in the universe of discoulp. This immediately implies that the information model is consistent as it has a model. The final information model is also assumed to have the properties that it is finite and that all perceived abstract knowledge about the universe of discourse is represented.

4. CHANGING AN INFORMATION MODEL

Assume that a partial information model O1 is obtained, which is changed and another one O2, is then obtained.

Four types of changes can be considered:

- a predicate symbol is added
- a predicate symbol is excluded
- a sentence is added (without changing the language)
- a sentence is excluded (without changing the language)

As pointed out above we assume that the information model O1 is satisfied in the considered universe of discourse.

4.1 Addition of a predicate symbol

When a predicate symbol is added to the language of the information model this will have no formal implications on the information model. However, from the point of interpretation it implies that a type of states-of-affairs is considered in the "new" information model O2 that was not considered in O1. In order to only represent the implied extension of the universe of discourse of the information model a tautological sentence can be added.

Example: Assume that also employees are considered in the universe of discourse, then the tautological sentence

$$\forall x(emp(x) \dashrightarrow emp(x))$$

can be added.

An addition of tautological sentences to an information model adds no knowledge to it as a tautological sentence is satisfied in all universes of discourse.

4.2 Exclusion of predicate symbols.

When a predicate symbol of an information model is excluded it implies that a type of state-of-affairs in the universe of discourse is disconsidered. When a predicate symbol is excluded from the information model also those sentences that include the excluded predicate symbol must be excluded, or changed. Then three cases can occur:

a) tautological sentences are excluded, this will not influence the remaining information model (cf. above).

b) sentences defined over the excluded predicate symbol only are excluded, neither this case will influence the remaining information model as the sentences are independent of the rest of the sentences. The resulting information model is satisfied in the universe of discourse but can become less informative than the original information model (see also section 4.4.).

c) sentences including the excluded predicate symbol are exluded. This case will be discussed in section 4.4.

4.3 Addition of sentences.

When a sentence e is added to an information model O1, giving O2, three cases can occur (cf. Lun-82b).

a) It holds that -e is deducible from O1
This means that if e is added to O1, the extended information model, O2, is inconsistent. This implies that O2 is not satisfied in any universe of discourse and in particular not in the considered one.

b) It holds that e is deducible from O1
Due to the soundness theorem for predicate logic this implies that the sentence e is satisfied in all models that satisfy O1, thus no additional knowledge is represented in O2 relative O1 and, thus, the sentence e is redundant in O2.

c) Neither e nor -e is deducible from O1
In this case the "new" information model O2 will be more informative than O1. But, this does not imply that it really exist a model for O2 that does not satisfy O1. This latter case will occur as soon as we have an information model that is not complete, e.g. when natural members are considered in the universe of discourse.

4.4 Exclusion of a sentence

Assuming that the information model O1 is satisfied in the universe of discourse and a sentence e is excluded then it can occur that O2 is less informative than O1. This follows directly as whenever O1 is satisfied in the universe of discourse so are all of its sentences and if one sentence is excluded the rest of sentences are still satisfied in the universe of discourse (cf. section 4.2). However, if the excluded sentence e is not deducible from O2, then O2 will be less informative than O1, i.e. e is not redundant in O1.

4.5. Discussion.

The main problem concerning the representation of information models in predicate logic is that of correctness checking. Assuming that the sentences of an information model are syntactical correct the only correctness criterion that can be applied is that of consistency. As we have pointed out above the consistency of an information model can be violated only when a sentence is added to it. For example, assume that a sentence is added to an information model and that the sentence does not violate the consistency. In such a case no formal criterion of correctness of the sentence with respect to the other sentences is available.

Example: Assume that the following information model is obtained:

$\forall x(secretary(x) \rightarrow emp(x))$

and assume that the following sentence is added

$\forall x(secretary(x) \rightarrow emp(x) \vee board\text{-}member(x))$

for the resulting information model it holds:
- the information model is consistent
- the second sentence is deducible from the the first sentence

But, should the second sentence replace the first sentence?

In order to increase the capability of determining the resulting information model after a change we have to find suitable meta-rules which can support a user when a model is changed. Consider the above example, in such a

case one could apply a rule saying that whenever the sentences include the same predicates then the sentences have to be intuitively inspected for correctness (i.e. with respect to the considered universe of discourse). However, such principles are outside the formal processing of information models.

5. RELATIVE STRENGHT OF INFORMATION MODEL

In what follows we will assume that the language of two information models, O1 and O2, are identical, i.e. they are defined over the same predicate symbols.

The information models are said to be equivalent, written O1=O2, if it holds for any sentence e that

O1 $\vdash$ e if and only if O2 $\vdash$ e

Further, we say that O2 is (deductively) stronger, or more informative, than O1, written O1>>O2, if for some sentence e it holds that

O2 $\vdash$ e and not O1 $\vdash$ e

We also say that an information model O is optimal if it does not exist a sentence e reflecting perceived abstract knowledge such that

not O $\vdash$ e

This should not be confused with the stronger concept of completeness of theories. Only for very simle cases it is possible to arrive at a complete theory (information model) and that is the reason for the weaker concept of optimality which refers to the perceived abstract knowledge. A theory is said to be monomorphic if it is consistent and all its models are isomorphic to each other, thus a monomorphic theory specifies all of the structural properties of its possible models (Car-54).

Further, an information model O is said to be non-redundant if for any sentence e in O it does not hold that e is deducible from O $-\{e\}$.

6. APPLICATIONS TO INFORMATION MODELS

In this chapter we will analyze and discuss some examples of constructs in information models with respect to their relative strength and its implications on the strenght of information models.

6.1 A simple example

Assume that we have the following concrete knowledge about a universe of discourse:

"Jim owns EMM300"

"John owns EUJ399"

"Jim owns LA6880"

An initial information model O1 is constructed:

$$O1=\{\forall x\forall y\forall z\forall u\forall v(own(x,y) \ \& \ own(x,z) \ \& \ own(x,u) \ \& \ own(x,v) \rightarrow (y=z) \ V \ (y=u) \ V \ (y=v) \ V \ (z=u) \ V \ (z=v) \ V \ (u=v))\}$$

Then a sentence e is added to O1, giving $O2=O1 \cup \{e\}$, where

$$e = \forall x\forall y\forall z\forall u(own(x,y) \ \& \ own(x,z) \ \& \ own(x,u) \rightarrow (y=z) \ V \ (y=u) \ V \ (z=u))$$

The relationships that hold for O1 and O2 are:

- $O1 >> O2$ as the sentence e can not be deduced from O1
- O2 is redundant as we can let $O2=\{e\}$
- we can say that O2 is optimal as it is assumed to reflect the perceived structural properties of the concrete knowledge above.

6.2 A typical example

Assume that we have the following abstract knowledge about a universe of discourse.

"all employees has a salary"

The predicates of the intended information model are:

exp(x) "x is an employee"

sal(x) "x is a salary"

esal(x,y) "employee x earns the salary y"

Assume that an initial information model is

$$O1 = \{\forall x(emp(x) \rightarrow \exists y(sal(y) \ \& \ esal(x,y)))\}$$

This information model can be made stronger as it permits any pair of entities to satisfy the predicate 'esal(x,y)'. Assume that a second information model is declared:

O2 = {Vx(emp(x) --> y(sal(y) & esal(x,y)))
VxVy(esal(x,y) --> emp(x) & sal(y))}

This information model is stronger than O1 as it restrics the instances for 'esal' to be pairs of entities where the first entity is an employee and the second is a salary. We say that we "close" the predicate 'esal' by the second formula. Further, the added sentence is not deducible from O1, i.e. a closing of the predicates makes the information model stronger.

The two formulae of O2 are typical instances of constructs of most approaches to information modelling. The first formula reflects a so called totality (total function (Bub-80)) which states that all instances of a set of entities are related in a particular way to other entities. The second formula represents a so called domain declaration which states that the entities of an association type are of certain types.

6.3 Relative strength of constructs.

The information model O2 of the preceeding section is a quite typical instance of information models as it expresses properties of all the instances of a set of entities. However, it should be noted that it is not assumed that it exists either exployees, salaries or esal-associations. In the next section we will discuss a possible assumption of existence of extensions of predicates. Here we will consider some general patterns of constructs and discuss their relative strength.

In most basic textbooks on predicate logic four basic constructs are presented and discussed; they are of the types:

∀x(trucker(x) --> employee(x))

∃x(trucker(x) & employee(x))

∀x(trucker(x) & employee(x))

∃x(trucker(x) --> employee(x))

The first construct implies the fourth, which is a weak construct. The fourth construct is satisfied in all universes of discourse except those in which, e.g., it exists truckers and all of them are not employees. This

type of construct will hardly be useful in practical cases. The third type of construct is very strong as it states something about all entities in the universe of discourse, which probably will be to strong in practical cases. Of more interest are the constructs of the first and second types.

First of all, we have to point out that the two first formulae above are incomparable in that neither of them are deducible from the other.

Let us consider the first formula, which states something about the instances of a set of entities. This type of construct has earlier been used in several approaches to information modelling. Examples are the totality constructs and domain declarations, which were discussed in section 6.2.

An alternative of the general pattern of the first formula is:

∀x(secretary(x) V trucker(x) --> employee(x))

This formula is stronger than, say

∀x(secretary(x) --> employee(x))

as the former states that those who are secretaries or truckers are employees and the latter states that secretaries are employees. Further, the former formula is reducible into

∀x(secretary(x) --> employees(x))

∀x(trucker(x) --> employees(x))

A weaker formula is

∀x(secretary(x) & trucker(x) --> employee(x))

which states that those who are both secretaries and truckers are employees. This formula is not reducible.

Correspondingly, we find that, e.g.

∀x(employee(x) --> secretary(x) V trucker(x))

∀x(employee(x) --> secretary(x) & trucker(x))

are weaker respectively stronger than, say

∀x(employee(x) --> secretary(x))

which is one of reductions of the implication with a conjunction in the consequent.

From these examples we conclude that in practical cases we should identify abstract knowledge that holds for as large as possible sets of entities and for which as many as possible properties hold. In particular, we have to "close" the predicates as much as possible (cf. section 6.5.).

Now, let us consider the second type of constructs. The basic construct is, say,

$\exists x(\text{secretary}(x))$

This construct is of course weaker than

$\exists x(\text{secretary}(x) \ \& \ \text{trucker}(x))$

as the latter states that it exists an entity that is both a secretary and a trucker. A weaker formula is, say,

$\exists x(\text{secretary}(x) \ V \ \text{trucker}(x))$

which states that is exists an entity that is a secretary or a trucker.
From these examples we conclude that one should if possible state that it exists entities which have a combination of properties. This is still more obvious if we combine the principal types of constructs, e.g.

$\forall x(\text{secretary}(x) \dashrightarrow \text{employee}(x))$

$\exists x(\text{secretary}(x))$

The first formula does not imply the existence of a secretary and then not the existence of an employee. If the second formula is also declared in an information model it immediately follows that it also exists employees.

6.4 Discussion on an existence assumption.

In the preceeding section we pointed out that one should if possible declare in an information model that it exists entities with certain properties. In this section we will discuss the implications of an assumption which states that every predicate has an instance. One can easily identify arguments supporting such an assumption, but also arguments against it.

An argument supporting the existence assumption is:

- when an information model is constructed and a predicate symbol is employed it exists a reason for including the predicate symbol, i.e. in the universe of discourse it exists state-of-affairs which are referred to by the predicate symbol.

However, a universe of discourse refers to a number of states of a portion of the real world and the existence assumption would then imply that in every state it exists an instance of the predicate symbols. This argument can easily be refused by assuming that the predicates include a variable which refers to states. Then, the existence assumption would be restated to: "it exists a state such that it exists ...". But, this is only a reduction of the problem and the same argument holds against the existence assumption in this case. Thus, we conclude that from a pure theoretical point of view an existence assumption should not be made. But, from a practical point of view it should be made as long as its advantages and disadvantages are considered.

Let us illustrate the advantage of the existence assumption by an example:

Assume that the following abstract knowledge holds for a universe of discourse:

"all who has a salary are employees"

"nobody is both an employee and a shareholder"

"every shareholder has a salary".

This is represented as follows:

$\forall x \forall y(esal(x,y) \rightarrow emp(x))$

$- \exists x(emp(x) \ \& \ sh(x))$

$\forall x(sh(x) \rightarrow \exists y(esal(x,y)))$

This set of formulae is consistent. However, from intuitive considerations it is concluded that it must be inconsistent. But, this is due to the implicit assumption that it really exists a shareholder. If this is assumed and included among the formulae above we will immediately have an inconsistent set of formulae. Thus, in this case the existence assumption is advantegous.

The following examples shows that one has to be aware of the disadvantages of the existence assumption. Assume that for a universe of discourse the following abstract knowledge holds:

"if there is a secretary there are no truckers"

"if there is a trucker there are no secretaries"

This is represented as follows:

$\exists x(secretary(x)) \dashrightarrow \forall x(\neg trucker(x))$

This formula is consistent. If the existence assumption is made then we have to add the following formulae:

$\exists x \exists y(secretary(x) \ \& \ trucker(y))$

Then, the extended set of formulae will immediately become inconsistent as the only universe of discourse that satisfies the original sentences is that in which all entities are secretaries or all entities are truckers, i.e. either of the sets of entities must be empty. Thus, in this case it was a disadvantage to assume the existence of an instance of the predicates.

However, the above case can be easily avoided by making the idea of states of the universe of discourse explicit. The abstract knowledge can then be restated as follows

"if there is a secretary in a state then there is no trucker"

which is represented as follows

$\forall x(\exists y(secretary(y,x)) \rightarrow \forall y(\neg trucker(y,x)))$

and the existence assumption gives

$\exists x \exists y \exists z \exists u(secretary(x,y) \ \& \ trucker(z,u))$

With this approach we avoid an inconsistency of the information model.

From a practical point of view we conclude that the existence assumption is advantageous as:

- it makes the information model stronger, (cf. section 6.3)
- it makes it more probable to find an inconsistency (cf. above)
- existence of entities are usually assumed in every-day reasoning (cf. above)

However, we have to be careful with the existence assumption when, as above, disjoint sets of entities are considered and, in particular, as in the last example, mutually exclusive sets of entities are considered.

6.5 Closing an information model

In section 6.2 we introduced the idea of closing a predicate which was examplified by a so called domain declaration, e.g.

$\forall x \forall y(esal(x,y) \rightarrow emp(x) \ \& \ sal(y))$

The formula states, e.g., that those objects that have a salary are employees. However, this is all that the formula represents. When such a formula is represented in an information model it is usually implicitly assumed that the objects that have a salary are employees, and are of no other types, such as shareholders. In order to obtain a stronger information model we should also state, if possible in the actual case, that the objects that have salaries are not also shareholders. This means that it should be stated both what "holds" and what "does not hold" for a set of entities. Thus, we can complete the above formula with the following:

$\neg \exists x \exists y(esal(x,y) \ \& \ shareholder(x))$

which is equivalent to

$\forall x \forall y(esal(x,y) \rightarrow \neg \ shareholder(x))$

However, this is not equivalent with that the sets of employees and shareholders are disjoint as an entity can be an employee who does not have a salary.

The idea of closing the predicates of an information model is quite similar to the closed world assumption of data bases, cf. (Rei-81). In our context the assumption implies, in principle, that an information model can be made complete (in the logical sense). However, as we pointed out above, we can not expect an information model to be complete (cf. chapter 5), but there are good reasons for aiming at "complete" (or, optional) information models:

- an information model becomes stronger

- implicit assumptions are made explicit.

These arguments imply that correctness checking of an information model will be more efficient, in particular consistency checking (cf. section 6.4).

7. CONCLUSIONS

In this paper we have focused on the process of changing an information model in order to obtain an information model that is as precise as possible with respect to the considered universe of discourse. Some principles to be applied in the construction of information models in order to obtain the above goal are presented. These principles include that the employed predicate symbols of an information model should be "closed" such that their instances are completely characterized. Further, it is proposed that an existence assumption about instances of the predicates should be made, though it has a limitation, which is pointed out. The relative strength of some typical constructs of information models are also discussed and examplified.

REFERENCES

Bub-80 Bubenko, Janis, jr.: "Information modeling in the context of system development", IFIP-80, Tokyo, Japan, 1980.

Car-54 Carnap, R.: "Einfurung in die symbolische Logik, mit besonderer Berucksichtigung ihrer Anwendung", Wien, Austria, 1954.

Hou-79 Housel, B.C., Waddle, V., Yao, S.B.: "The functional dependency model for logical database design", IBM Res.Lab., San Jose, USA, 1979.

Iso-82 Griethuysen, J.J. (ed): "Concepts and Terminology for the Conceptual Schema and the Information Base", ISO TC97/SC5/WG3, 1982.

Lun-82a Lundberg, B.: Contributions to Information Modelling, Ph.D.-thesis, Stockholm, Sweden, 1982.

Lun-82b Lundberg, B.: "IMT - an Information Modelling Tool", IFIP WG 8.1 Work.Conf. on Automated Tools for Information Systems Design and Development, New Orleans, USA, 1982.

Rei-81 Reiter, R.: "Data Bases: A logical perspective", SIGMOD, Vol 11:2, 1981.

Sen-77 Senko, M.E.: "Conceptual schemas, abstract data structures, enterprise descriptions", Internat. Comp. Symp., 1977.

METHODOLOGY FOR THE REPRESENTATION OF SOFTWARE PRODUCTION PROCESSES

M. De Blasi and G. Turco
Istituto di Scienze dell'Informazione,
Universita' di Bari, Italy

1.INTRODUCTION

Software production processes are measurable entities, as are software products themselves. They also include software products, but their essential components are production activities. In this study, "production activity" is taken to mean the sum total of manual and automatic operations required to pass from one product to another.

Among production activities, even the measurements are to be

considered. In this case, two levels of analysis are established: the first refers to the object to which the measurement activity is applied and the other to the activity itself.

The production process is an entity in development, not terminated as on the contrary is a software product. The analysis of a production process is directed towards the knowledge of objects which must still be produced, with the aim of influencing production mechanisms themselves.

Elshoff /1/ uses complexity as a control variable in the production process: the programmers are constantly supplied with feedback on the code which they are producing, so that when it becomes too complex, they are asked to reprogram it until values of acceptable complexity are obtained.

Belady and Lehman /2/ see the large software systems as organisms which change during their lifetime in relation to their environment.

We maintain the necessity of intensifying the interactions with the environment during the initial stages and decreasing them after the product has been released. This may be done by means of measurements, as in Elshoff, which generate a feedback on the process itself.

The analysis of production processes must have previsional characteristics in relation to the product to be obtained. De Millo and Lipton /3/ suggest that ideas should be taken from less precise sciences than Physics -for example Meteorology or Economics-, since in these, as in Software Science, the predictive component is much more important than the explanatory one.

The summary of the Panel Findings of /4/ states that: "A natural

dichotomy exists in the interests of those who study software metrics. There are those whose interests lie in studies of the creation and management of programs - in human performance. And there are those whose interests lie in studies of the object produced - in program performance. Although it is generally agreed that there ought to be a natural relationship between these two types of studies, we see no unifying theory developing in the near future".

So, software products and production processes should be represented in the same space and analytical relations between the former and the latter should be established.

Belady /5/ also maintains that it is difficult to develop a metric for both products and processes. Many experiments (see for example Sayward /6/) have been conducted to measure products and few to measure processes. Sayward also reports various experiments which relate the two areas. However, these suffer from the lack of a unifying theory based on a univocal product and production process representation.

This study introduces a definition space for production processes. In this space, a production process corresponds to a trajectory made up of segments representing the activities. The final and intermediate points correspond to the various products obtained during the entire process. Or else they may be isolated from the trajectory in order to represent products which already exist, and can be used in the production process.

Section 2 deals with the definition space of software production processes.

In the Sections following, the representation of processes and products in this space, its metric basis and usage of analytical relations as previsional and comparing tools, are developed.

An application of the methodology is carried out on some production processes studied in a preceeding paper /7/.

The tools for production process measurement, introduced in /7/, are described in detail in /8/, where a presentation is made of an application in an industrial environment of the methodology proposed for the production of large scale software.

2. DEFINITION SPACE OF SOFTWARE PRODUCTION PROCESSES

A software production process can be represented by a trajectory of a point moving through a space, whose coordinates are measurable properties such as: production time, programming cost, functional requirements, execution time, memory occupation, level of portability, maintenance level, readability, complexity, etc.

According to the production tools available (hardware and software) and the preselected strategies, we will have various trajectories and various arrival points of these trajectories. The choice of which trajectory to follow should take into account those arrival points found within a predetermined area of the definition space ("area of acceptability").

Intermediate points may also influence the choice of optimum trajectory. Assuming that time is a privileged variable, it may be interesting, for example, to determine trajectories which connect products, P(t), obtained at different times, ti and tf, with equal functional characteristics, but with different performances, whose final points, P(tf), are of course still within the area of acceptability ("prototyping" and "tuning").

In certain cases, it may be preferable to follow this kind of trajectory, instead of one which has a final point with greater characteristics, but no intermediate points of the type described. In fact, in this way, we have the advantage that, from the first phases of software generation, a product is already obtained with the required functionality, even if the other prerequisites are not yet satisfied. This is useful for testing and evaluation purposes.

Groups of variables in this space may be part of particular metric bases, according to which aspects of product or production process are enphasized in the analysis. Among the most important, we indicate the well-known metrics based on the analysis of the program text (Halstead /9/) and those, complementary to them, which are based on its history of execution (Knuth /10/).

The software product continues its trajectory even after what we have called its arrival point. Actually it is at this point that it begins to exist as a "finished product". From this moment on, other variables become important, specifically: costs of maintenance, transport if any, modification, extension, error elimination, execution time,

occupation of memory and of other resources.

It is essential that the entire trajectory, and not only the point determining the final product, is situated, from this moment onward, in an area of acceptability, in order to ensure that the quality requirements, whether set or forecast, will be constantly satisfied.

Many quantities possess this double aspect which refers to "before" and "after" the product has been obtained. That is, there is one cost for preventive therapy and one for the intervention on the product. For example, Jones /11/ separates quality measurements into measures of defect removal efficiency and defect prevention.

Thus, the conclusion reached is that the trajectory should be chosen according to the arrival point of the product and the intermediate points, and also according to "future" points. How to choose a trajectory on the basis of measurements of a product which still must begin to exist, is a problem which, within other sciences, is solved in two possible ways:

a) making use of simulators which, by underlining determined characteristics each time, also allow their measurement and thus the evaluation and choice of the trajectory which optimizes that partial set of characteristics;

b) determining other variables, from which "future" variables may be deduced by means of analytical relations. This is equivalent to increasing the dimensions of the definition space, in order to incorporate these new and fundamental coordinates.

Both methods must generate 1) an adequate instrumentation for the

measurements of the quantities of interest, and 2) a set of analytical relations and/or invariance principles for the interpretation of measurements carried out. The difference between the two methods lies in the varying importance assumed by the two above mentioned points.

Each time one proceeds towards such a modelization, introducing groups of variables characterizing each aspect of a software product or of a production process, with tools for their measurements and analytical relations for their interpretation, a definition is made of a "Physics" or, better still, of a branch of Software Physics.

Returning to the concept of a trajectory in a definition space of software products, it is worthwhile focusing attention, for a moment, on particular types of trajectories: those which join two points of two distinct trajectories, as in the conversion of a product from one computer to another.

In this operation, the most obvious variable is the cost of transport, which may vary to a great extent, according to the level of portability of the original software. Moreover, if this is not portable, there are two groups of trajectories, whether or not the arrival product is portable, thus causing a notable difference in transport cost. The importance of this variable is so great as to wrongly overshadow other factors such as: time efficiency, memory occupation, level of maintainability, etc., thus limiting us to a simple maintenance or generic improvement of values assumed in the original product. Also in this case, a physical approach cannot avoid the examination and measurement of all variables on which software product trajectories

depend, so as to forecast the characteristics and performance of the final product, thereby carrying out the choice of the trajectory which achieves the best compromise.

Tools used in the software production, such as languages, compilers, interpreters, code generators and operating systems, are software products as well. The characteristics of products to develop and thus the various trajectories are dependent on them. Generally, software production tools correspond to isolated points in space, since they are almost always products already obtained, supplied by the firm or by an external software manufacturer.

In other cases, production tools correspond to final trajectory points, if it is the user who must produce them. Testing and debugging tools and precompilers, are examples appearing in this category. Generally, we may safely say that the measurement tools themselves are to be measured and evaluated in the production space.

Furthermore, it is often the case that, even if products supplied by an outside manufacturer are involved, they lack evaluation in terms of even such basic figures as execution time and the like, so that, in order to carry out our analysis, it is necessary to have the proper measurement tool available for these products as well.

As far as the analytical relations existing between production process variables are concerned, we have already seen their previsional properties with regard to characteristics of products still to be obtained. We have also insisted on the importance of comparing the various trajectories with one another, not only on the basis of puntiform

characteristics, but also of continuous intervals of the variables. Thus, it is important not only to obtain analytical relations between the variables of a production process, that is, relations along a single trajectory, but also to obtain analytical relations between the trajectories, eventually taking one of them as a trajectory of reference and relating the others to it.

3. MODELS

As examples of software production processes, we take the ones studied in /7/. The following is a brief description of the corresponding models.

Hypotheses made in /7/ on the environment included:

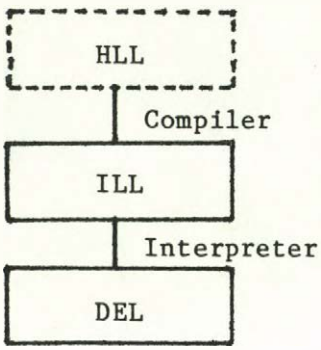
- the existence of three language levels: HLL (High Level Language), ILL (Intermediate Level Language) and DEL (Directly Executable Language), with no reference made to the particular languages used;
- the use of normal production tools, among which in particular there were both an interpreter and a code generator from ILL to DEL;
- the use of "tuning" methodologies.

On the basis of the above, five alternative models were formulated:

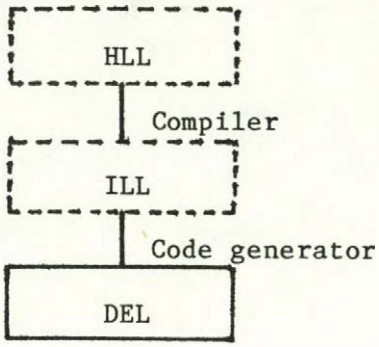
1) DEL model, consistent in the writing in DEL (or in the symbolic correspondent): it defines a machine at one level (Fig. 1.1);

2) The "interpretive" model (Fig. 1.2), which concerns the writing of

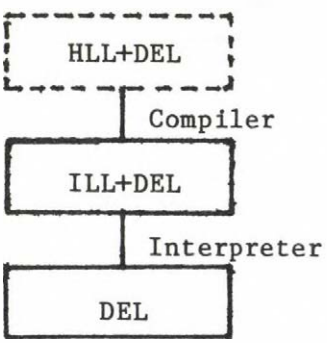
1) DEL model



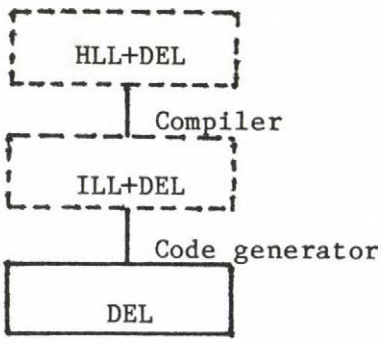
2) Interpretive model



3) Generative model



4) Interpretive +tuning model



5) Generative +tuning model

Fig.1 Software production models

programs HLL which, for reasons of portability, are compiled in ILL and then interpreted in DEL;

3) The "generative" model (Fig. 1.3), which differs from the interpretive model in that it uses a code generator to pass from ILL to DEL;

4) The interpretive model with tuning (Fig. 1.4);

5) The generative model with tuning (Fig. 1.5).

Models 4 and 5 are respectively models 2 and 3 optimized in execution time. The tuning methodology is applied by measuring the critical HLL areas and substituting them with DEL code.

4. PRODUCTION TRAJECTORIES

An observation which may be made, before moving on to the application of considerations made in the preceeding Sections to models introduced, is that, in order to represent entities (activities and products) in a definition space of production processes, their metric basis should be defined. On the other hand, this metric may only be deduced by analysis of activities involved in the different production processes, so that it is preferable to follow the order of first introducing the production trajectories - referring to "production time" - and then, in the following Sections, the coordinates and analytical relations essential for their analysis.

Some definitions are given and then "elementary" trajectories are

introduced.

A trajectory, within the definition space, is composed, as has already been stated, of segments and vertices: the former indicating production activities and the latter, the products. A segment always goes from one product to another, or rather always joins two vertices. The first segment also begins from a "product", supplied by the sum total of the initial unformalized specifications of the product to be obtained.

We shall follow the convention of labelling only the vertices - not necessarily all of them - , reporting, for the sake of brevity, the languages in which the products are obtained.

The first kind of trajectory that we shall consider, concerns measurement activities. Let us take, for example, the activity of counting n , the number of instructions executed in a given DEL program run. The corresponding diagram is shown in Fig. 2. It represents a measurement activity which leads to the passage from DEL product to DEL:n product.

The two usual ways of carrying out measurement, using instrumentation or interpretation, are represented respectively in Fig.3 and in Fig.4.

Interpretation activities are denoted by dashed segments, in order to indicate the fact that they are not activities of transformation from one product to another, but rather of execution of a product on a machine whose language is expressed by the second vertex. The measurement of n in Fig.4 is seen as a minor variation in the interpretation activity.

Another kind of trajectory is found in "conversion" diagrams, which

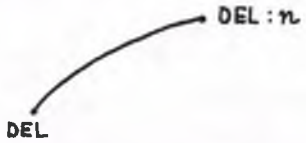


Fig.2 Measurement trajectory of n

Fig.3 Measurement of n by means of DEL instrumentation

Fig.4 Measurement of n by means of interpretation

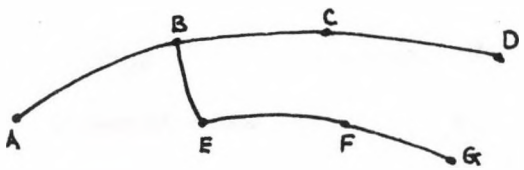


Fig.5 Diagram of conversion from machine D to machine G

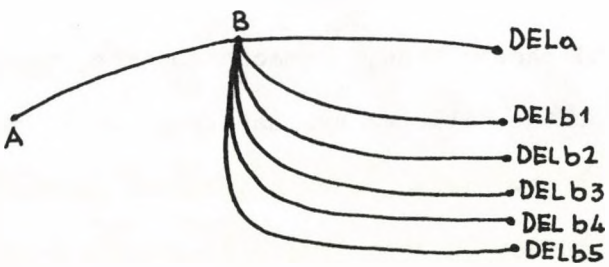


Fig.6 Diagram of conversion of a DELa product to a DELb machine, examining five different production processes, leading to products all functionally identical.

lead to the passage from a trajectory to one or more different trajectories. An example of a conversion diagram is supplied in Fig.5, which represents the case of two products, functionally identical, but implemented on different machines. In this case, the two processes share only the initial analysis activity, while they diverge in the final production activities.

In a more general way, conversion diagrams may be interpreted as the representation of altenative production processes obtaining functionally identical products. Such processes may have some sections in common, in both the initial and final parts, supplying different products in one case and the same product in the other. The diversification of final products does not necessarily mean the use of different machines, but much more often different procedures making use of the same hardware.

A problem of conversion from a DELa to a DELb machine, which examines various alternatives, such as those outlined in the preceeding Section, may be represented schematically as in Fig.6.

The sum total of initial activities, common to the various trajectories, has no influence on the relative evaluation of the various production processes. Thus, this evaluation may include the production process followed to obtain the product to be transported, or else it may be limited only to alternative processes.

In any case, the diagram of conversion is transformed into one of "selection" between various independent alternatives. Fig.7 traces the selection diagram for the five models introduced in the preceeding

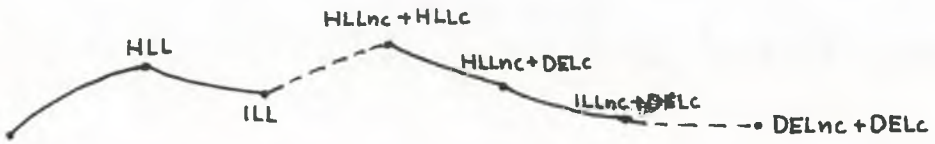
Section, and Tab.1 lists the corresponding products.

(1) DEL writing. Reference trajectory.

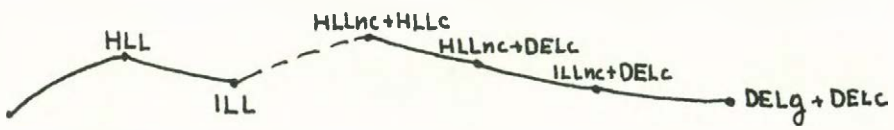
(2) HLL writing, ILL compilation, interpretation.

(3) HLL writing, ILL compilation, DEL code generation.

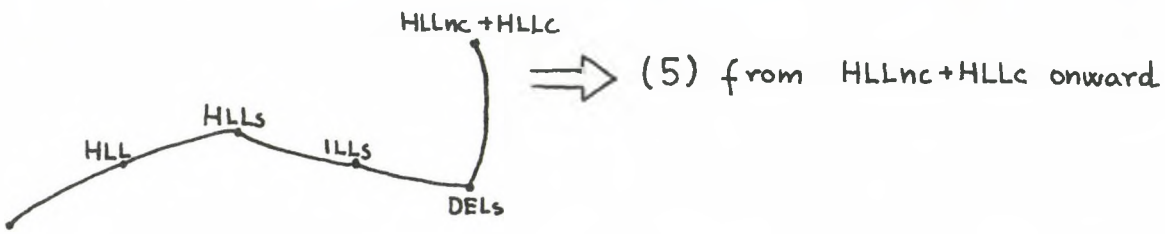
Fig.7 Production trajectories for: 1) DEL, 2) Interpretive, and 3) Generative models.



(4) HLL writing, ILL compilation, interpretation with HLLc measurement, rewriting of HLLc in DEL, ILL compilation, interpretation.



(5) This is identical to (4), except for the latter activity which involves DEL code generation.



(5′) HLL writing, HLLs instrumentation, ILL compilation, DEL code generation, execution to obtain HLLc, then (5), from HLLnc+HLLc onward.

Fig.7 (Cont.) Production trajectories for: 4) Interpretive + tuning, 5) and 5′) Generative + tuning models.

Table 1. SOFTWARE PRODUCT LIST

Product	Definition
a) DEL	Program written in Directly Executable Language
b) HLL	Program written in High Level Language
c) ILL	Program in Intermediate Level Language, obtained by compilation from HLL
d) DELg	Program in Directly Executable Language, obtained by code generation from ILL
e) HLLnc+HLLc	Program in High Level Language measured to detect the critical areas HLLc
f) DELc	Critical areas rewritten in Directly Executable Language
g) HLLs	Program in High Level Language instrumented to detect critical areas
h) ILLs	Program in Intermediate Level Language obtained by compilation from HLLs
i) DELs	Program in Directly Executable Language, obtained by code generation from ILLs
j) ILL/DEL	Interpreter of Intermediate Level Language in a Directly Executable Language machine
k) ILL/DELg	Code generator from Intermediate Level Language to Directly Executable Language.

The trajectories of Fig. 7.1, 7.2 and 7.3 reflect the simplicity of the corresponding models. It is only worth noting that there is a diversity of products obtained by the first and the third process, from which the different notations, DEL and DELg, are taken. As can be readily seen, there is a difference not only in the text of the two programs, but also in their properties, as a result of the different DEL and DELg language levels. This is evident from measurements of ILL power carried out in relation to both of them (/7,8/).

The trajectories related to models with tuning (Fig. 7.4, 7.5 and 7.5′) are increased by the insertion of measurement activities.

In Fig. 7.4 and 7.5, after obtaining the product in ILL, measurement is carried out of the critical areas, by means of a modified interpreter. The product obtained is the knowledge of the critical part HLLc. The measurement activity in both production processes is then followed by the rewriting of HLLc in DELc.

It is noteworthy that DELc is to be considered different from DEL, since the rewriting is less expensive than the writing activity.

Rewriting is followed by integration and compilation in ILL. Lastly, the two processes are differentiated by the final activities of interpretation and code generation respectively.

A variation of the trajectory in Fig. 7.5, indicated in Fig. 7.5′, involves the use of code generation from the initial activities onward. To allow for this, an HLL instrumentation is used, followed by code generation and compilation, according to the HLLs, ILLs and DELs chain. The instrumented program is then executed to determine the critical areas

HLLc: it is at this point that the trajectory in Fig. 7.5′ is converted to that of Fig. 7.5 for the final activities of DELg+DELc production.

5. METRIC BASIS

Now, for each product of Table 1, the variables which characterize it are to be determined in the light of the theory which is being constructed. That is, a definition must be made of all the variables (and only those) from whose value one can predict the characteristics of the final products, by means of suitable analytical relations. This ensures that the system of variables is complete and minimum.

In the construction of the system of variables, a useful nucleus for starting is supplied by precisely those variables which characterize the final products. It is assumed that the only variables of interest for these products are those which define that we have called AREA OF ACCEPTABILITY.

For example, in /7/, the proposal was made to choose, from the possible production processes, those for which the following remained within predetermined limits of acceptability:

C, the production cost;

T, the execution time;

O, the memory occupation; and

P, the portability,

on the understanding that working methods which guarantee maintainability, reliability, readability and expandability, would in any case be followed. Table 2a summarizes these variables.

This first set of variables is then enlarged with variables which define intermediate products and can influence the properties of final products.

These are, above all, the variables related to the METHODOLOGY. In tuning methodology, we define (Table 2b):

$$p = T(HLLc)/T(HLL) \text{ , and}$$

$$r = O(HLLc)/O(HLL) \text{ ,}$$

i.e. the fractions of execution time and of memory occupation of critical areas, denoted here by HLLc, with respect to the whole program HLL. They are measured by the activities which produce HLLnc+HLLc in Fig. 7.4, 7.5 and 7.5'.

Another set of variables is realated to the TOOLS used in the production processes. In our case, the tools which have a notable influence on the final products are the interpreter ILL/DEL and the code generator ILL/DELg (products j and k in Table 1). The quantities defined for these products also characterize the languages between which they operate, i.e. ILL and DEL.

In Table 2c these variables are defined. They are the 'interpretation cost' and the dynamic and static 'powers' of ILL with

respect to both DEL and DELg. All these depend on the relative characteristics of the languages involved as well as on the tools used.

Lastly, some variables are concerned with the LANGUAGES alone. Generally, they are instruction execution times and instruction lengths.

Table 2. THE METRIC BASIS

a) Variables defining the area of acceptability:

C PRODUCTION COST

It includes cost of programming, debugging and testing activities. On the contrary, the costs of automatic activities, such as compilation, code generation and measurements of critical areas, are neglected compared with the above. C(DEL), C(HLL) and C(DELc) are, thus, the only costs to be measured.

T EXECUTION TIME

O MEMORY OCCUPATION

P PORTABILITY

b) Variables related to the methodology:

r = O(HLLc)/O(HLL)

Fraction of the HLL program to be rewritten in DEL, resulting from the measurement activity of the critical HLL areas.

p = T(HLLc)/T(HLL)

Fraction of time spent in critical areas.

c) Variables related to the tools:

N INTERPRETATION COST OF ILL

It is the mean number of DEL instructions that the interpreter carries out to extract, examine and execute an ILL instruction.

Mt POWER OF ILL

It is given by the mean number of DEL instructions which would carry out the same operation as a single ILL instruction. It can be measured considering two "equivalent" programs in DEL and in ILL, i.e. a program written in DEL and another in HLL (compiled in ILL) for the same problem. By dividing the number of instructions executed in each of the two programs one obtains the power of ILL.

Ms STATIC POWER OF ILL

The same as Mt, except that it is given by the ratio of the lengths of the two programs.

M't APPARENT POWER OF ILL

It is the power of ILL calculated with respect to DELg, i.e. it is the mean number of instructions executed in the generated program for carrying out the operation of a single ILL instruction.

M's APPARENT STATIC POWER OF ILL

Static power of ILL with respect to DELg. Both M't and M's are greater than Mt and Ms respectively. In other words, ILL,

when related to its DELg, seems more powerful than it really is.

d) Variables related to the languages:

k DEL INSTRUCTION EXECUTION TIME

l(DEL) DEL INSTRUCTION LENGTH

l(ILL) ILL INSTRUCTION LENGTH

All these quantities are to be taken as weighted averages.

6. ANALYTICAL RELATIONS

The variables C, T, O and P, which define the final products, are related to the other variables introduced in Section 5 by means of a number of formulae which we have grouped in Table 3. These expressions furnish the values of C, T, O and P, relative to those of process 1 or DEL process, except for the portability.

The deduction of these relations is given in /7/. Here, however, we illustrate their use as previsional tools and in the comparison of different processes.

Table 3. ANALYTICAL RELATIONS

C1 = C(DEL)

C21 = C(HLL)/C(DEL)

C31 = C(HLL)/C(DEL)

C41 = C21 + r = C(HLL)/C(DEL) + r

C51 = C31 + r = C(HLL)/C(DEL) + r

T1 = T(DEL)

T21 = N/Mt

T31 = M't/Mt

T41 = (1-p)T21 + p = (1-p)N/Mt + p

T51 = (1-p)T31 + p = (1-p)M't/Mt + p

O1 = O(DEL)

O21 = 1(ILL)/(1(DEL)Ms)

O31 = M's/Ms

O41 = (1-r)O21 + r = (1-r)1(ILL)/(1(DEL)Ms) + r

O51 = (1-r)O31 + r = (1-r)M's/Ms + r

P1 = 0

P2 = 1

P3 = 1

P4 = 1-r

P5 = 1-r

As far as the costs are concerned, as a consequence of that which has been asserted in Table 2a, we consider only the writing activities, i.e. programming in HLL, in DEL and reprogramming critical areas in DEL.

As can be seen in Table 3, the relative costs all depend on the ratio:

$$C(HLL)/C(DEL) \ .$$

This is a constant characteristic of the production environment which can be easily measured.

The last two costs refer to the processes adopting the tuning methodology. Their expression:

$$C41 = C51 = C(HLL)/C(DEL) + r \ , \qquad (1)$$

is a linear function of r, i.e. of the fraction of code which we decide to rewrite.

The variables r and p are of course related to one another (see Knuth /10/). That is, if we reprogram a small fraction of code, we will have a low cost, but likewise we will have little improvement in execution time. In quantitative terms, this can be evaluated from the expressions reported in Table 3 for the relative times using tuning:

$$T41 = (1-p)T21 + p$$

$$T51 = (1-p)T31 + p \ , \qquad (2)$$

which establish a relationship between the times without tuning and those

with tuning. The quantity (1-p) can be defined as the 'time reduction factor' for processes which adopt tuning.

Thus, the expressions (2) can be used together with (1) for deciding an optimal choice of the parameters p and r, which obtain the maximum improvement in execution time remaining, at the same time, within acceptable costs.

Passing to the terms T21 and T31, it is seen in Table 3 that these can be expressed by means of more 'fundamental' variables:

$$T21 = N/Mt$$

$$T31 = M't/Mt$$

i.e. by means of the ILL interpretation cost and the ILL powers. This gives us the possibility of an "a priori" evaluation of the times. In addition, the two processes, interpretive and generative, can be compared with one another, by remembering, from Table 2c, the definitions of N and M't. From these one has:

$$M't < N$$

and, then:

$$T31 < T21$$

and

$$T51 < T41 \quad .$$

Again, only by taking measurements on the tools used, we have obtained quantitative statement of the known property that affirms that the code generated is more efficient in execution time than the code interpreted. Indeed, from these expressions, we can also say how much more efficient the former is than the latter. A quantitative knowledge is always essential, if a trade-off has to be reached between various performance requirements.

We can make analogous considerations on the memory occupations, except that static quantities are to be taken into account.

Finally, we see from Table 3 that the portability is also quantified, using tuning. Thus, the choice of the size of critical areas to reprogram has to be made also taking into account this variable.

7. CONCLUSIONS

We believe that it is always possible, in each production environment, to represent both the software products and the software production processes in the same definition space with a restricted number of coordinates. This has been demonstrated for the environment studied in this paper.

We have constantly tried to reach generality. Abstraction from our environment lead us to obtaining a metric basis and a system of analytical relations. These can constitute, we hope, a starting basis for

development of a theory incorporating instances from other environments.

REFERENCES

/1/ J.L.Elshoff, "A Review of Software Measurement Studies at General Motors Research Laboratories", Proceedings of the Second Software Life Cycle Management Workshop, 166-171, IEEE, New York, 1978

/2/ L.A.Belady and M.M.Lehman, "The Characteristics of Large Systems", Research Directions in Software Technology, 106-138, MIT Press, Cambridge, Massachusetts, 1979

/3/ R.A.DeMillo and R.J.Lipton, "Software Project Forecasting", in F.G.Sayward, M.Shaw and A.J.Perlis, editor, Software Metrics: An Analysis and Evaluation, 77-94, MIT Press, Cambridge, Massachusetts, 1981

/4/ F.G.Sayward, M.Shaw and A.J.Perlis, editor, "Sofware Metrics: An Analysis and Evaluation", MIT Press, Cambridge, Massachusetts, 1981

/5/ L.A.Belady, "Software Complexity", Software Phenomenology, 371-383, AIRMICS, Atlanta, 1977

/6/ F.G.Sayward, "Design of Software Experiments", in F.G.Sayward, M.Shaw and A.J.Perlis, editor, Software Metrics: An Analysis and Evaluation, 43-59, MIT Press, Cambridge, Massachusetts, 1981

/7/ M.DeBlasi, D.Marino, O.Murro, "Une methodologie pour l'evaluation de strategies de production de logiciel", Actes des Journees BIGRE 82, 113-127, Grenoble, 1982

/8/ M.Carulli, C.Marzano, V.Tetro, G.Turco, "Valutazione delle strategie di produzione del software: risultati di una metodologia", Atti AICA, 53-59, Padova 1982

/9/ M.H.Halstead, "Elements of Software Science", New York, Elsevier, 1977

/10/ D.E.Knuth, "An Empirical Study of FORTRAN Programs", Software-Practice and Experience, Vol.1, 105-133, 1971

/11/ T.C.Jones, "Measuring Programming Quality and Productivity", IBM System Journal 17 (1), 1978.

The Development of a Machine Independent Multi Language Compiler System Applying the Vienna Development Method

Uwe Schmidt Reinhard Völler

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel

1 Project Background and Motivation

Since 1980 the Computer Science Department of Kiel University and Dietz Computer Systems, Mülheim, FRG, have been cooperating in the development of a uniform compiler system for the languages BASIC, COBOL, FORTRAN and PASCAL, supported by Dietz.

A main requirement was the easy portability of the system to new computers and a high code efficiency, because the source languages are used for systems programming and CAD applications.

For this purpose a machine independent high level intermediate language was derived from formal denotational semantics specifications of the source languages. In this language CAT (**C**ommon **A**bstract **T**ree Language) programs are represented as abstract program trees. CAT is especially suited for the implementation of the four languages mentioned above, but other languages can be compiled into CAT as well. If necessary new constructs can be added.

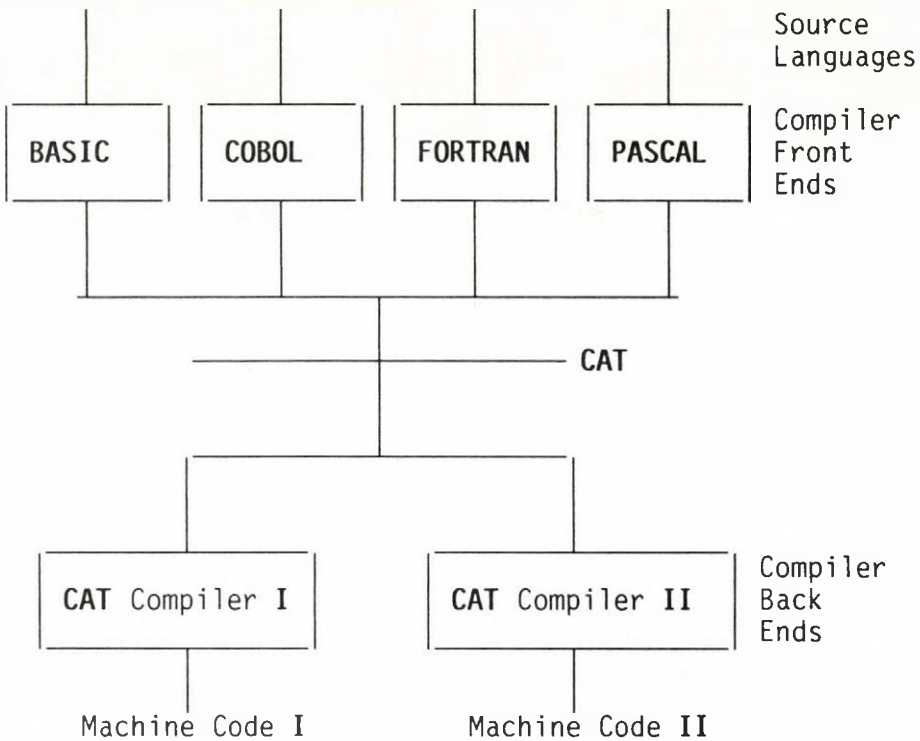
The specifications of the compiler front ends are derived from the specifications of the dynamic semantics of the source languages. This approach assures the correctness of the compilation process.

The front ends can be kept relatively simple, because of the high level of the intermediate language. Input for the front ends is an abstract program tree constructed by a parser, which is automatically generated. The machine independence of CAT guarantees the portability of the system.

The target machines and the respective machine languages are formally specified in a uniform manner. These specifications lead to compiler back ends, which convert the trees of the CAT language into sequences of machine instructions. The conversion of the back end specifications is to a large extent done automatically.

The Vienna Development Method (VDM) [1] is used for the specification of all languages and compiler front and back ends. This formal method automatically yields complete and provably consistent specifications, which can systematically and in part automatically be transformed into high level language programs.

The following diagram gives a general view of the entire system:



2 The Development of the Intermediate Language

2.1 The Need for Formal Language Definitions

The existing language definitions for BASIC, COBOL, FORTRAN and PASCAL are all informally written in natural language and thus leave room for different interpretations. To eliminate the ambiguities and inconsistencies of these definitions, we extract formal denotational specifications from the language standards and manuals [2,3,4,5].

A further advantage of this approach is that now the comparizon of different languages is made considerably easier. The formalization of language descriptions helps to detect constructs, which are semantically but not syntactically equivalent, as well as constructs with the same syntax but different semantics. This approach was also recently chosen in a language comparizon of CHILL and Ada [6,7].

The language used for these descriptions is META IV, the specification language of the Vienna Development Method. The META IV specifications consist of an abstract syntax, which abstracts the strings defined by the concrete syntax of a language to mathematically tractable objects, such as trees, sets and maps. Additional contextual dependencies of the syntactic objects are formulated through the "is-well-formed"- functions of the static semantics. Interpretation functions of the dynamic semantics associate objects from semantic domains, which consist mainly of a storage model, to the syntactic objects. These descriptions form the basis for the derivation of the intermediate language and the development of the compiler front ends. They are automatically complete and their consistency is provable.

2.2 The Derivation of CAT

The derivation of the intermediate language is based on the interpretation functions defining the dynamic semantics of the various language constructs. In order to identify constructs with common semantics, the interpretation functions use the same semantic domains [8].

2.2.1 The Common Storage Model

The most important of these domains are the domains needed for the storage model. Without such a common storage model the interpretation functions are not comparable. The complexity of this model varies with the different source languages.

Since PASCAL allows references to values of structured variables, a model of structured values of array and record variables is necessary. This model must also reflect the changes of a value of a record variable caused by an assignment to a component in a variant part.

The model must be general enough to cope with the effects caused by FORTRAN **EQUIVALENCE** statements and COBOL **REDEFINES** and **RENAMES** clauses. These constructs permit the implicit redefinition of overlayed variables. Therefore information about the relative position in store and the length of a value is needed. The programmer can define the storage layout of data using the **COMMON** and **EQUIVALENCE** statements in FORTRAN and thus exploit the side effects of assignments to equivalenced variables. The same applies to COBOL data records.

We model a value as a map from pairs of natural numbers to elementary values:

Val = (Rad Freead)⟶ Simpleval
Simpleval = **INTG** | **REAL** | ...
Rad = **NO**
Freead = **NO**

Addresses are pairs of locations and relative addresses:

Locval = (Loc Rad)

Rad denotes a relative address, Freead the first unused address. In the environment we keep information about the offsets of record fields.

The following META IV function specifies the updating of a storage cell:

```
1. 1   store(loc,rad,len,vm)(stg) =
 . 2      if loc not ϵ dom stg
 . 3      then error
 . 4      else
 . 5       let ovm = stg(loc)
 . 6          ,chg-locs =  (i,j) | ((i,j) ϵ dom ovm) &
 . 7                                ({i:j-1} ∩ {rad:rad+len-1} ≠ {})
 . 8          ,(min, ) ϵ chg-locs s.t.
 . 9            (A (i,j) ϵ chg-locs)(i ≠ min ⇒ i > min)
 .10          ,( ,max) ϵ chg-locs s.t.
 .11            (A (i,j) ϵ chg-locs)(j ≠ max ⇒ j < max)
 .12          ,ovm' = (ovm - chg-locs) +
 .13                  [(k,k+1) :-  UNDEF | (min <= k <= rad) v
 .14                                       (rad+len <= k < max)]
 .15          ,nvm  = ovm' + [(i+rad,j+rad) ↦  vm((i,j)) |
 .16                          (i,j) ϵ dom vm]
 .17       in stg + [loc ↦  nvm]
 .18   type: Loc Rad Len Val ⟶ Stg ⟶ Stg
```

The location referenced must be a valid address (.2-.3). The old value is read and that part of the value computed, which overlaps with the new value (.4-.10). Components, which only partially overlap are set to an undefined value (.11-.13). Finally the value is updated and a new storage returned. The function for reading a value from store is similar:

```
2. 1   read(loc,rad,el)(stg) =
 . 2      if loc not ϵ dom stg
 . 3      then error
 . 4      else
 . 5       let vm = stg(loc)
```

```
. 6          ,v = [(i-rad,j-rad) ↦ vm((i,j)) |
. 7                ((i,j) ϵ dom vm) & ({i:j} c {rad:rad+el})]
. 8       in if (union { {i:j-1} | (i,j) ϵ dom v } ≠ {0..el-1})
. 9          then error
.10          else v
.11   type: Loc Rad NO ⟶ Stg ⟶ Val
```

The value to be read is specified by a location, an offset within the location and a length (.1). The entire value is read and those components extracted, which are selected by the offset rad and the length el (.4-.6). If the value is not well-formed an error occurs.

Having defined a common storage model for the four languages, we now turn to the identification of the syntactic constructs to be included in the intermediate language.

2.2.2 The Elements of the Intermediate Language CAT

The syntactic objects necessary for a common intermediate language are chosen from the union of all syntactic objects of the source languages according to the following criteria:

- From language elements which have the same semantics in several source languages only one is chosen for the intermediate language.
- If a syntactic object of one language is the general case of one or more objects of other languages, then only this element is included in the intermediate language.

An example is the **loop**-statement, we included in CAT. It implements the PASCAL **repeat**-, **while**- and **loop**-statements and is also used for the implementation of the FORTRAN **DO**- and the PASCAL **for**-statement. Its abstract syntax and dynamic semantics are:

```
Loop            :: s-ini  : Statement
                   s-exit : Expr
                   s-fin  : Statement

3. 1  i-loop(mk-Loop(s1,c,s2),env)(stg) =
. 2     let f(fstg) = (let stg1 = i-statement(s1,env)(fstg)
. 3                       ,(mk-Tv(v1,  ),stg2)
. 4                            = e-bool(c,env)(stg1)
. 5                    in if v1 = 0
. 6                       then f(i-statement(s2,env)(stg2))
. 7                       else stg2)
. 8     in f(stg)
. 9   type: Loop Env ⟶ Stg ⟶ Stg
```

This construct implements a **while**-loop, when s1 is the empty statement and a **repeat**-statement, when s2 is empty.

- Complex objects which can be broken down into a sequence of simple objects already in the intermediate language are ommitted.

The **for**-statement from PASCAL and the **DO**-statement from FORTRAN can be broken down into a series of assignments, a test and a loop-statement and are therefore not included in the intermediate language.

- Finally new objects are defined, which implement several other constructs of the source languages.

We may take the exception handling facility in CAT as an example. This construct is similar to the exception mechanism in Ada and can be used for the implementation of global jumps in PASCAL, **ON ERROR** conditions in COBOL, exception handlers in BASIC and the handling of runtime errors.

It is necessary to reach a compromise between the size of the intermediate language and the complexity of the compiler front ends. The front ends can be kept simple through the direct inclusion of language constructs. This should always be done

for constructs present in several languages. However this leads to a large intermediate language to be handled by the compiler back ends. A small intermediate language on the other hand requires that the front ends must perform relatively complex transformations. This could lead to unnecessary duplicate work. A construct should not be broken down into instructions, which later have to be recombined by the compiler back ends.

The approach taken assures the completeness of CAT, because only constructs which are implementable by means of other language elements are not directly included in CAT.

The abstract syntax and the static and dynamic semantics of the included objects yield a formal specification of the intermediate language.

2.3 The Specification of the Compile Algorithms

The partial evaluation and the rewriting of the interpretation functions lead to transformation functions mapping the different language elements to syntactic constructs of the intermediate language. They form the specification of the compiler front ends.

The information in the environment is known at compile time and can be used in the partial evaluation of the interpretation functions. However the information contained in the storage is only known at runtime. Therefore every time a storage access is performed or a state transformation made, code has to be generated, which performs the state transition at execution time. The following functions give an impression of the relation between the interpretation and compilation functions:

```
4. 1   i-repeat-st(mk-Repeat-st(c,st),env)(stg) =
 . 2      let f(fstg) =
 . 3           let fstg1 = i-statement(st,env)(fstg)
 . 4              ,(fstg2, mk-Tv(v, )
 . 5                    = e-bool(c,env)(fstg1)
 . 6           in if v = 0
 . 7              then f(fstg2)
 . 8              else fstg2
 . 9      in f(stg)
.10   type: Repeat-st Env ⟶ Stg ⟶ Stg

5. 1   c-repeat-st(mk-Repeat-st(c,st),env) =
 . 2      let loop =
 . 3           let cst = c-statement(st,env)
 . 4              ,cc  = c-bool(c,env)
 . 5           in mk-Loop(cst,cc,NIL)
 . 6      in loop
 . 7   type: Repeat-st Env ⟶ Loop
```

The resulting specifications are then transformed by hand into executable PASCAL code.

3 The Development of the CAT Compilers

The CAT compilers of different target machines are all of the same structure and are developed following a uniform approach [9].

First a simple, universal, low level language CAL (CAT Assembly Language) is defined.

3.1 The CAT Assembly Language

Syntactic Domains of CAL

```
1. 1   CAL-program    =   Instr *
 . 2   Instr          =   Assign  |  Branch  |  Jump  |  Label
 . 3   Assign         ::  Var  Expr
 . 4   Branch         ::  Expr  Labelname
 . 5   Jump           ::  Expr
 . 6   Label          ::  Labelname

 . 7   Expr           =   Const  |  Var  |  Operation
 . 8   Const          ::  Mode  Val
 . 9   Var            ::  Mode  Rad  Base  [Ix]
 .10   Operation      ::  Mode  Opcode  Expr *
 .11   Base,Ix        =   Expr
 .12   Rad            =   INTG

 .13   Val            =   INTG  |  "all other CAT values"
 .14   Mode           =   Subr-mode  |  "all other CAT modes"
 .15   Subr-mode      ::  Lb  Ub
 .16   Lb,Ub          =   INTG
```

A program in the CAT Assembly Language consists of a sequence of instructions (.1), which is interpeted sequentially. There are only four kinds of instructions: assignments, conditional and unconditional jumps and labels (.2).

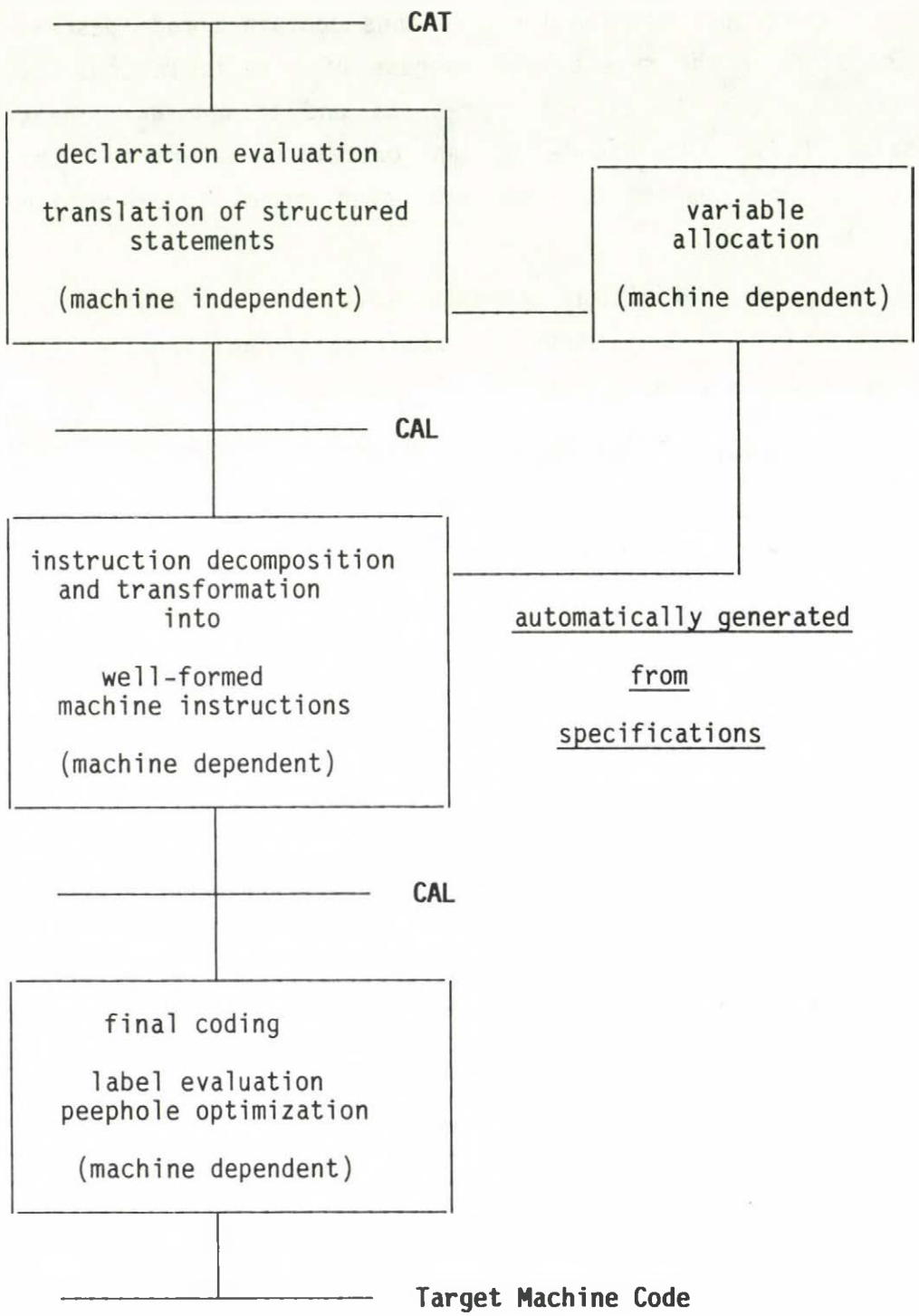
The components of an assignment are a variable as destination and an expression as source (.3). Components of a conditional jump are a boolean expression and a label (.4). The destination of an unconditional jump is determined by an expression (.5).

We distinguish three kinds of expressions: constants, variables and operations (.7). All expressions contain a mode describing the kind of the result. The address of a variable consists of a constant offset, a base expression and an optional index expression (.9). The opcode of an operation determines the function to be applied to the evaluated argument expression list (.10).

The recursive definition of expressions allows arbitrarily complex operations and address expressions. The domains for values and modes are taken from the CAT language.

3.2 The Structure of the CAT Compilers

The translation of CAT into a concrete machine language is performed in three steps. The following diagram illustrates the structure of the compilers:



First all CAT language elements are mapped to corresponding CAL instructions for an "ideal" CAT machine, that is a machine, which directly implements the operators and functions used in CAT. This step includes the resolving of control structures, the declaration evaluation and the storage allocation. Except for the storage allocation this step is machine independent and identical for all compilers.

The second step is the real machine code generation phase. In this phase opcodes of the target machine are substituted for CAT opcodes with the same semantics. Complex instructions are broken down into sequences of simpler instructions. The necessary temporary allocation is performed by the variable allocation routines.

The quality of the generated code is determined by this second phase and the variable allocation scheme. The development is based on a formal description of the available machine instructions.

A final simple step generates the concrete bit pattern of the instructions required by the target machine. Here the label evaluation and optionally a peephole optimization is done.

3.3 The Machine Descriptions

The description of a specific target machine contains the context conditions for the instructions and the meaning functions for the available opcodes.

The context conditions are given by a set of "is-well-formed"-functions (similar to the specification of the source languages), describing the limitations operands are subject to on the concrete target machine (one-, two- address instructions, register instructions).

Basis for the formal description are the informal descriptions in the manufacturers' manuals.

Example: specification of addition instructions for one of the target machines (National 16000 processor).

opcodes	operation description and operand restrictions
ADD	$d := e_1 + e_2$ - addition of signed integers - overflow test - first operand e_1 = destination d - d general address - e_2 general operand
ADDQ	$d := e_1 + c_2$ - addition of signed integers - overflow test - first operand e_1 = destination d - d general address - c_2 integer constant in $\{-8..7\}$
INDEX	$d := (e_1 * (e_2 + 1)) + e_3$ - arithmetic with nonnegative integers - no overflow test - $d = e_1$ - d, e_1 general purpose registers - e_2, e_3 general operands

The following excerpt of the "is-wf-move"- function shows the formal version of these context conditions for "add"- instructions.

```
1. 1   is-wf-move(d,s) =
 . 2     let mk-Operation(m,op,el) = s in
 . 3     cases op :
 . 4      (ADD      →    el = ⟨e1,e2⟩
 . 5                   & d = e1
 . 6                   & is-general-addr(d)
 . 7                   & is-general-opnd(e2)
 . 8      ,ADDQ     →    el = ⟨e1,e2⟩
 . 9                   & d = e1
.10                    & e2 = mk-Const(m2,v2) & v2 ∈ {-8..7}
.11      ,INDEX    →    el = ⟨e1,e2,e3⟩
.12                    & d = e1
.13                    & is-gpr-addr(d)
.14                    & is-general-opnd(e2)
.15                    & is-general-opnd(e3)
.16        :
.17        :
.18      )
.19   type :  Var  Expr  ⟶  BOOL
```

The functions is-general-addr, is-general-opnd and is-gpr-addr are the predicates to be met by the operands.

3.4 The Derivation of the Codegenerators

From the predicate function "is-wf-move" a routine "gen-wf-move" is constructed which transforms the instructions generated in the first machine independent compiler phase into a sequence of well-formed target machine instructions.

First the opcodes of the "ideal" CAT machine are replaced by opcodes of the target machine. The replacement of the integer addition operator $+_i$ by National 16000 opcodes is shown in the following excerpt of "gen-wf-move":

```
2. 1   gen-wf-move(d,s) =
 . 2     let mk-Operation(m,op,el) = s in
 . 3     cases op :
 . 4     (+i  →  let ⟨e1,e2⟩ = el in
 . 5             if {s-Lb(m)..s-Ub(m)} c {min-int..max-int}
 . 6              & e1 = mk-Operation(m1,*i,⟨e11,e12⟩)
 . 7              & s-Lb(s-Mode(e11)) ⩾ 0
 . 8              & s-Lb(s-Mode(e12)) > 0
 . 9              & s-Lb(s-Mode(e2)) ⩾ 0
 .10             then gen-wf-move(d,
 .11                    mk-Operation(m,INDEX,⟨e11,c-sub(e12,one),e2⟩))
 .12             else gen-wf-move(d,
 .13                    mk-Operation(m,
 .14                      if e2 = mk-Const(m2,v2) & v2 ϵ {-8..7}
 .15                      then ADDQ else ADD,
 .16                      ⟨e1,e2⟩))
 .17       :
 .18       :
 .19     )
 .20   type :  Var  Expr  ==⟩
```

First it is checked, whether the addition is implementable by an "INDEX"- instruction. It is tested that no overflow can occur (.5), that the first subexpression is a multiplication (.6) and that all operands are nonnegative (.7-.9). Otherwise the "ADDQ"- or "ADD"- opcode is selected. After the selection of the appropriate operators the operands are manipulated to meet the requirements of the context conditions. This may lead to the generation of additional instructions. The second excerpt of "gen-wf-move" demonstrates these transformations for the opcodes "ADD", "ADDQ" and "INDEX".

```
3. 1   gen-wf-move(d,s) =
 . 2     let mk-Operation(m,op,el) = s in
 . 3     cases op :
 . 4     (ADD,ADDQ  →
 . 5           let ⟨e1,e2⟩ = el in
 . 6           if e1 = d & is-general-addr(d)
 . 7           then if is-general-opnd(e2)
 . 8                then out-move(d,s)
 . 9                else gen-wf-move(d,
.10                       mk-Operation(m,op,⟨e1,
.11                         gen-general-opnd(e2)⟩))
.12           else
.13             let e22 = gen-indep-expr(e2,d)
.14                ,d1  = gen-general-addr(d)
.15             in (gen-wf-move(d1,e1)
.16                ;gen-wf-move(d1,mk-Operation(m,op,⟨d1,e22⟩)))
.17     ,INDEX  →
.18           let ⟨e1,e2,e3⟩ = el in
.19           if is-gpr-addr(d)
.20           then if e1 = d
.21                then if is-general-opnd(e2) & is-general-opnd(e3)
.22                     then out-move(d,s)
.23                     else gen-wf-move(d,
.24                            mk-Operation(m,op,
.25                              ⟨e1,gen-general-opnd(e2),
.26                                 gen-general-opnd(e3)⟩))
.27                else (gen-wf-move(d,e1)
.28                     ;gen-wf-move(d,mk-Operation(m,op,⟨d,e2,e3⟩)))
.29           else gen-wf-move(d,gen-gpr-addr(s))
.30       :
.31     )
.32   type :  Var  Expr  ==⟩
```

In this process the operands are transformed to successively fulfill the "is-wf"- predicates. For the "ADD"- and "ADDQ"-instructions first the conditions for the destination d and the first operand e_1 are tested (.6). If these predicates are met, the second operand e_2 is checked (.7) and the instruction is emitted or a new simpler second operand is computed (.9-.11). If $d \neq e_1$ or d is not accessable e_2 is simplified to make the evaluation of e_2 independent of d (.13) and d is made accessable (.14). Now code for the evaluation of e_1 in d and the addition is generated (.15-.16).

The "gen-wf-..."- functions are associated with the "**is**-wf-..."- predicates that the following equation holds for all expressions e :

is-wf-...(gen-wf-...(e)) = **true**

The following function "gen-gpr-addr" for evaluation of an expression in a general purpose register is a characteristic example for the "gen-wf-..."- functions:

```
4. 1   gen-gpr-addr(e) =
. 2      if is-gpr-addr(e)
. 3      then e
. 4      else (def t : alloc-gpr(s-Mode(e))
. 5           ;gen-wf-move(t,e)
. 6           ;return t)
. 7   type :  Expr  ==> Expr
```

In contrast to the implementation of the compiler front ends the codegenerators are automatically generated from the specifications written in a subset of META IV without manual interference. This prevents errors introduced by a manual conversion of the formal discriptions into executable programs.

4 Project Status

For three years two people have been involved in this project. A PASCAL-CAT compiler front end and two CAT compiler back ends for DIETZ-621 and National 16000 have been implemented. The entire system is written in PASCAL and running on a DIETZ-621 with 2 * 40 Kbyte of storage. The quality of the generated code is comparable to good handcoded assembly language programs. It has turned out that the code runs up to 20% faster than the code generated by the PASCAL compiler, which is up to now used for systems programming by DIETZ. The implementation of a CAT compiler for a new target machine requires an effort of approximately six man months. There are no restrictions to the applicability of our approach to existing conventional processors.

The specifications of the compiler front ends for the four source languages are complete (with the exception of COBOL where only the translation of the data structures was specified). The CAT definition is a document of approximately 3000 lines of META IV formulae. The PASCAL specification has about the same size. It took about six man months to extract a formal specification for FORTRAN 77 from the ANSI- standard. This excludes the FORTRAN I/O and runtime support. The compile algorithm for PASCAL consists of 1600 lines of META IV corresponding to a 6000 line PASCAL program. The specifications of the code generators for the DIETZ-621 and the National 16000 processor have a size of about 3000 lines each. They are written in a restricted subset of META IV and can be compiled into PASCAL programs.

The correctness of the development of a code generator is supported considerably by the application of the Vienna Development Method [10,11]. Costly design errore are avoided. It has been shown that a formal method like VDM can be applied in a real life industry project with good success. Without this tool it would not have been possible to develop such a universal system in such a short period of time.

5 References

[1] Bjorner, D., Jones, C.B., The Vienna Development Method: The Meta-Language, Berlin, Springer, LNCS 61, 1978

[2] Jensen, K., Wirth, N., PASCAL User Manual and Report Berlin, Springer, LNCS 18, 1974

[3] ANSI X3.9, American National Standard Programming Language FORTRAN, New York, American National Standards Institute, 1978

[4] ANSI X3.23, American National Standard Programming Language COBOL, New York, American National Standards Institute, 1974

[5] ANSI X3.J2, American National Standard Programming Language BASIC, Draft Proposal, New York, American National Standards Institute, 1980

[6] Meiling, E., Palm, S.U., A Comparative Study of CHILL and Ada on the Basis of Denotational Descriptions, DDC 66/1 982-12-31, Dansk Datamatik Center, Lyngby, 1982

[7] Meiling, E., Palm, S.U., A Storage and Environment Model for CHILL and Ada, DDC 66/1982-12-24, Dansk Datamatik Center, Lyngby, 1982

[8] Schmidt, U., Völler, R., Die formale Entwicklung der maschinenunabhängigen Zwischensprache CAT, GI - 11. Jahrestagung, Berlin, Springer, Informatik-Fachberichte 50, 1981

[9] Schmidt, U., Völler, R., Die Entwicklung eines portablen Übersetzersystems mit der Vienna Development Method, Implementierung PASCAL-artiger Programmiersprachen, Berichte des German Chapter of the ACM 11, B. G. Teubner, Stuttgart 1982

[10] Bjorner, D., The Systematic Development of a Compiling Algorithm, Techn. Rept. ID681, Dept. of Comp. Sci., Techn. University of Denmark, Kopenhagen, 1977

[11] Bjorner, D., Programming Languages: Formal Development of Interpreters and Compilers, in Morlet, E., Ribbens, D., International Computing Symposium 1977, Proceedings, Amsterdam, North-Holland Publ. Comp., 1977, p. 1

A System Model for Vertical and Orthogonal Migration*

B.Holtkamp, H. Kaestner
University of Dortmund, Informatik III
Postfach 500500
D-4600 Dortmund 50, F.R.Germany

1 Introduction

Vertical and orthogonal (or outboard) migration are well-known techniques to improve the performance of a computing system seen as a hierarchy of software/firmware/hardware. To apply both techniques the following steps have to be performed:

1. identification of suitable candidates (system components) to be migrated,

2. prediction of results (performance improvements) that can be expected,

3. implementation of the components selected in step 1 and 2,

4. verification of the system's behaviour after the migration process with respect to the results of step 2.

Changing the implementation environment of a system's component (i.e. migrating this component either vertical or outboard) needs a careful investigation of the component's interconnections. This can be done best if there is a modelling tool by which the relevant structures of the real system can be described.

With regard to vertical migration such system models have been evaluated ([STO 78], [STA 81], [DAV 83]). For a combined approach to both vertical and orthogonal migration a different model is needed. It has to allow the description of parallel processes which are most important for orthogonal migration.

In this paper we present a system model which fulfills the above requirement. It is exactly described in the next chapter. In chapter 3 we discuss structural constrains for migration candidates in terms of our system model. Thus it is demonstrated how the model serves as a base for the migration steps described above. Chapter 4 shows how the structural aspects discussed so far can be combined with data on the dynamic system behaviour to give a framework for migration step 2.

* This work is partially supported by DFG (Deutsche Forschungsgemeinschaft) under contract Ri 367/2-1

2 System Model

The migration system model that is introduced in this chapter allows the abstract description of systems to which the migration technique is applied. It helps to identify the candidates to be migrated and to investigate structural requirements for them.

There are two areas which influenced our model. The first one is related to the concrete structure of basic software in a computing system. According to [LAU 78] operating systems and real-time systems can be constructed in two ways:

1. using a procedure-oriented approach

2. using a process-oriented approach

Both approaches are supported by modern systems implementation languages like ADA [LED 81] and Modula-2 [WIR 80]. Consequently our system model also has to allow the description of such structures.

The second area is related to the hardware support for orthogonal migration. For our purposes we assume a hardware system having attached one or more coprocessors to the same system bus. These systems are further distinguished according to the coprocessors' ability to access main memory or not.

With these two areas in mind we describe our model in a top-down manner.

A system is defined as
S = (SGS, SAL, SPS, SOS, SAS)
with

SGS is the system's global state space

SAL is the system's access list (defined at the end of this chapter)

SPS is the system's procedure set

SOS is the system's object set (possibly empty)

SAS is the system's action sequence which is performed when the system is initialized.

Objects are optional so that programs to be implemented in languages like PASCAL or C can be directly modelled.

The concept of objects serves two purposes. First it allows to define the components of a system by means of a set of operations. These operations are the only ones which can manipulate the internal representation of the component thus preserving its invariant properties [JON 78]. By adding one or more object managers to each object [JAM 77] along with an appropriate set of actions, process systems can be modelled.

Formally an object is defined as follows:
$O_i = (O_iN, O_iSS, O_iAL, O_iOS, O_iPS, O_iMS, O_iAS)$
with

O_iN is the name of object i

O_iSS is the state space of object i

O_iAL is the access list of object i which defines the accesses that may be performed from outside the object and to its environment (see the end of this chapter)

O_iOS is the object set of object i, i.e. the set of (nested) objects local to the current object

O_iPS is the procedure set of object i

O_iMS is the object manager set of object i

O_iAS is the action sequence of object i which is performed when the object is initialized.

An object manager has the following components:
$M_{ij} = (M_{ij}N, M_{ij}SS, M_{ij}PS, M_{ij}AS)$
with

$M_{ij}N$ is the name of object manager j belonging to object i

$M_{ij}SS$ is the state space of the object manager j belonging to object i

$M_{ij}PS$ is the procedure set of object manager j belonging to object i

$M_{ij}AS$ is the action sequence of object manager j belonging to object i

In terms of our definition a procedure looks similar to an object manager:
$P_k = (XP_kN, XP_kFPL, XP_kSS, XP_kPS, XP_kAS)$
with

XP_kN is the name of procedure k within X = (S or O_i or M_{ij})

XP_kFPL is the formal parameter list of procedure k within X

XP_kSS is the state space of the procedure k within X

XP_kPS is a set of procedures local to procedure k within X

XP_kAS is the action sequence of procedure k within X which is performed each time the procedure is called

The main difference between an object manager and a procedure is with regard to the actions which may be used and the relation to their environment.

An action sequence describes the transformations to be performed either on the local or the surrounding global data space:
$AS = (a_1,\ldots,a_n)$
with
a_i : xSS ---> xSS

Each a_i is an element of the set of elementary actions which are defined for our model. We distinguish between normal actions (denoted by A) and special actions which are relevant for vertical and orthogonal migration. First of all there is the procedure call denoted by:
pcall = (XP_kN, XP_kAPL)
with

XP_kN is the name of the called procedure

XP_kAPL is the list of actual parameters

To describe the actions of processes we follow the concept of "synchronizing resources" [AND 81]. The operations which may be performed on an object have to be defined in the in actions within the object manager:
in = (o_1 ,..., o_m)
where each o_1 describes an operation which is defined within the in action.

Each operation has the form:
$o_1 = (o_1N, o_1BE, o_1FPL, o_1IR, o_1SE, o_1AS)$
with

o_1N is the name of operation 1

o_1BE is a Boolean expression

o_1FPL is the formal parameter list of operation 1

o_1IR describe the invocation restrictions (either call or send)

o_1SE is a scheduling expression

o_1AS is an action sequence.

The name of the operation and the Boolean expression constitute a guard [DIJ 75]. The guard is true if at least one pending invocation of the named operation and the corresponding Boolean expression is true. If there is more than one pending invocation, these are ordered by increasing values of the associated scheduling expression (if this is omitted, the order is unde-

fined).

Execution of an in action proceeds as follows. If at least one of the guards is true, an arbitrary one is chosen. The first of the pending invocations of the associated operation is selected and the action sequence is executed. If no guard is true, the in action is delayed until at least one of them becomes true. The in action terminates when one of the operations has been executed.

For the invocation of operations there are two actions call and send :

call = (o_1N, o_1APL), send = (o_1N, o_1APL)

with

o_1N is the name of an operation to be executed

o_1APL is the list of actual parameters supplied for the invocation.

If the operation is invoked by call, the invoking object manager is delayed until the operation has been executed by the object manager supplying it within an in action. If on the other hand invocation is by send, the invoking object manager may continue its actions as soon as the actual parameters (message) have been transmitted.

Objects, object managers and procedures each represent a natural border around the data structures and operations respectively, which are defined in the corresponding state space (xSS) or within an in action. There are two principal ways in which these borders can be crossed. The first is an implicite one and holds for the following conditions:

a: A procedure nested within another procedure may directly access the local state space (XP_kSS) of the surrounding procedure.

b: A procedure with an object or object manager may directly access the O SS of the surrounding object or the $M_{ij}SS$ and O_iSS of the surrounding object manager and the corresponding object.

These conditions correspond to the scope rules of traditional block structured languages. They are not valid, however, for objects. For this reason we introduce the notion of an object access list:

O_iAL = (O_iDOL, O_iDDL, O_iUOL, O_iUDL)

with

O_iDOL is a list of operations which are defined within the included object manager and which may be invoked from outside

O_iDDL describes a subset of O_iSS which may be accessed from nested objects

O_iUOL is a list of operations defined in DOLs of other objects and invoked by the object manager of this object

O_iUDL describes those data structures of an outer object or the system which are accessed by this object.

The system access list SAS has the same form as the component O_iDDL.

This completes the presentation of our system model. In the next chapter we will use it to discuss structural aspects of vertical and orthogonal migration.

3 Structural Aspects of Migration

The system model which has been formally introduced in the previous chapter will now be used to discuss structural aspects of both vertical and orthogonal migration. For these purposes we define some metrics on those components that can be migrated.

3.1 Migration Metrics

As we have pointed out in the introduction one needs to know the exact interaction of a migration candidate with its environment. In terms of our system model the components object, procedure, and object manager are possible candidates for migration. This means that for these candidates the interaction with the environment is specified by accesses to global data structures and by the execution of the actions pcall, call and send.

A convenient tool to describe the latter form of interaction is by means of call graphs. To model the aspect of data passing by means of these calls and the aspect of accessing global data structures, we will introduce some terms which are also used in the field of software structure metrics [HEN 81].

Definition 3.1:

a) The global data flow function gdf(P) of a procedure P is defined as the size of the global data structures, which P accesses in outer procedures or in the surrounding object manager or object plus the size of the externally accessed data structures (defined in the UDL of the corresponding object).

b) The local data flow function ldf(P) of a procedure P is defined as the size of all parameter lists contained in actions of type pcall within P.

c) The procedure call out-degree pcout(P) of a procedure P is defined as the number of actions of type pcall contained in P.

d) The procedure call in-degree pcin(P) of a procedure P is defined as the number of actions of type pcall outside P which call P.

In a similar way corresponding functions for object managers and objects can also be defined.

Definition 3.2:

a) The global data flow function gdf(M) of an object manager M is defined as the size of those data structures, which M accesses in the corresponding object, plus the size of the externally accessed data structures (defined in the UDL of the object).

b) The local data flow function ldf(M) of an object manager M is defined as the size of all parameter lists contained in actions of type call and send within M.

c) The send out-degree sout(M) of an object manager M is defined as the number of actions of type send contained in M.

d) The call out-degree cout(M) of an object manager M is defined as the number of actions of type call contained in M.

e) The send in-degree sin(M) of an object manager M is defined as the number of actions of type send outside M which invoke an operation contained in the DOL of the corresponding object.

f) The call in-degree cin(M) of an object manager M is defined as the number of actions of type call outside M which invoke an operation contained in the DOL of the corresponding object.

Definition 3.3:

a) The global data flow function gdf(O) of an object O is defined as the size of the data structures contained in the UDL.

b) The local data flow function ldf(O) of an object O is defined as ldf(M) of the included object manager.

The metrics defined above will now be used to describe structural aspects of migration.

3.2 Structural Aspects of Vertical Migration

According to the two principal structures of operating systems (see section 2 of chapter 2) we will discuss vertical migration for both of them separately.

In a procedure-oriented system we assume to have no objects. Thus we are only concerned with attributes of procedures.

Definition 3.4:

A procedure P is vertical migratable from a structural point of view if

a) pcout(P) = 0 or

b) all procedures contained in the call subgraph with root P are "migratable".

Part a of the definition is related to the fact that software functions cannot be called from the firmware. Thus only leave nodes in the call graph (part a) or complete subgraphs (part b) can be migrated.

Lemma
For two procedures P_1 and P_2 with $pcout(P_1) = pcout(P_2) = 0$. P_1 is a better candidate for vertical migration from a structural point of view than P_2 if one of the following conditions hold:

a) $gdf(P_1)$ $!(P_2)$

b) $PL(P_1) < PL(P_2)$

c) $pcin(P_1) > pcin(P_2)$

The background for this lemma is given by some hardware restrictions. The first one is that main memory references slow down the execution of microprograms and the second that the number of internal registers for local variables is limited.

The aspects discussed for procedure-oriented systems are also valid for process-oriented ones. This is because objects managers can also be seen as procedures as there is no parallelism between software and firmware.

3.3 Structural Aspects of Orthogonal Migration

For orthogonal migration we assume a system with objects. Depending on the coprocessor's local memory size, complete objects or some procedures (not necessarily contained in the same object) might be candidates for migration. With regard to single procedures there is no structural difference to vertical migration.

In hardware systems, where the coprocessor has no direct access to the main memory, all objects which have a non-empty UDL cannot be considered for orthogonal migration.

In general the main problem with orthogonal migration of objects is related to the communication structure between the object managers. As it makes no difference whether M_1 (running on the main processor) invokes an operation of M_2 (running on a coprocessor) or vice versa, invocation directions are not important. Instead we can concentrate on the communication relations of single boject managers.

Definition 3.5:

a) An object manager M is of define-type NIL, if it does not contain an action of type in.

b) An object manager M is of define-type call, if all operations in DOL are invoked by actions of type call.

c) An object manager M is of define-type send, if there is at least one operation in DOL which is invoked by an action of type send.

d) An object manager M is of use-type NIL, if UOL of the corresponding object is empty.

e) An object manager M is of use-type call, if all operations in UOL are invoked by actions of type call.

f) An object manager M is of use-type send, if at least one operation in UOL is invoked by an action of type send.

Definition 3.6:

An object O is of type t_1/t_2, if its object manager is of define-type t_1 and use-type t_2.

With these definitions we get the following type of objects (sorted according to decreasing suitability for orthogonal migration):

NIL/NIL These objects only make sense if they access global data structures (e.g. monitoring processes). In this case they are good candidates for orthogonal migration.

The following type group of objects contains either a define-type or use-type send or both. They represent suitable candidates for orthogonal migration because the send action activates a process (object manager) which may run in parallel, if the corresponding object is on a different (co)processor:

NIL/SEND
SEND/SEND
SEND/NIL
SEND/CALL
CALL/SEND

The remaining three types NIL/CALL, CALL/NIL, and CALL/CALL are no good candidates from a structural point of view, because they do not imply any parallelism.

Beside the type of an object the ldf(M) of its object manager is also important. It can be used to define a sequence between object managers of same type.

4 Quantitative Aspects of Migration

In this chapter we will introduce some cost functions with regard to migration candidates. Their purpose is to valuate the dynamic behaviour of the candidates and their hardware dependencies. They provide a base for the selection (step 2 of the migration technique) of migration candidates.

4.1 Quantitative Aspects of Vertical Migration

For the valuation of migration candidates we can differ between static and dynamic measures. As a general static parameter the control store space request is accepted (see [LUQ 80], [STO 78]).

So the static costs C_s for a procedure P are expressed by the following term:

$$C_s(P) = \sum_{i=1}^{n} m(a_i)$$

with $m(a_i)$ is the memory space for a_i and $a_i \in XAS$.

However, the static costs of a function are not a sufficient criterium for the selection of migration candidates. They must be weighted with the dynamic behaviour of a procedure with respect to the architectural characteristics of the processor.

As dynamic parameters execution frequency and average execution time are considered. Based on [PRY 82] execution time can be composed of instruction fetch and execute time (T_x) and the time for data references devided into global (T_g) and local (T_l) ones. The division of global and local data references seems to be necessary, because on one hand their access characteristics normaly differ and on the other hand the firmware level often provides more registers into which local data structures can be mapped. The dynamic costs of a procedure are described by the following equation:

$$C_d = n \times T$$

with n : execution frequency
T : average execution time

In order to ease the prediction of time savings for the migrated version of a function, the average execution time is splitted:

$$T = T_x + T_g + T_l$$

T_g and T_l can be written as

$$T_g = N_{gr} * t_{gr} + N_{gw} * t_{gw}$$

$$T_l = \sum_{i=1}^{c} (N_{lr}[i] * t_{lr}[i] + N_{lw}[i] * t_{lw}[i])$$

where the N denotes the number of global or local read and write accesses, respectively, and the corresponding t characterizes the time effort for such an access that depends on the addressing mode and processor speed. The references to local data are devided into different classes because various addressing modes can be used.

For the estimation of time savings the dynamic costs C_d, for the migrated version can be calculated follows:

$$C_{d'} = n * (T_{x'} + T_g + T_{l'})$$

How to calculate $T_{x'}$ and $T_{l'}$ is not further considered here, because it is not important for our model. What we can derive from the above equation is a weight W_p for a migration candidate:

$$W_p = (C_d - C_{d'})/C_s$$

While static measures can be calculated by a modified compiler, the only way to get information about the dynamic parameters is using a monitor. A well-suited tool for measuring the quantities mentioned above is described in [HOL 82]. The architectural parameters (t_{gr}, t_{gw}, t_{lr}, t_{lw}, number of registers on firmware level) are specified in processor manuals.

To perform the selection process structural and quantitative aspects can be combined in the following backtracking algorithm:

> In the first step all procedures with pcout(P) = 0 are sorted by decreasing weight W_p. They constitute the basic set of migration candidates.
> In a second and further steps those procedures are added, which only call members of the basic set. Their weights, which do not include the weights of the called functions, have to be corrected, i.e. the ldf and pcout will have an effect because intra level data and control transfers are moved from software to firmware level.
> Finally from this set those elements are taken which result in a control store filling with maximum weight.

4.2 Quantitative Aspects of Orthogonal Migration

Similar to our discussion on structural aspects we will first consider procedures as candidates for migration. The cost functions C_s and C_d of the previous section can also be applied for orthogonal migration. They now look like:

$C_s^{MP}(P)$ static costs for procedure P when implemented on the main processor

$C_s^{CP}(P)$ static costs for procedure P when implemented on a coprocessor

$C_d^{MP}(P)$ dynamic costs for procedure P when implemented on the main processor

$C_d^{CP}(P)$ dynamic costs for procedure P when implemented on a coprocessor

In terms of these functions a necessary condition for procedure P to be a migration candidate is:

$$C_d^{CP}(P) < C_d^{MP}(P)$$

For the second type of system, i.e. with objects, we will restrict ourselves to the migration of complete objects. Function C_s^x can be defined as above. For function C_d^x we will not take single actions as a measurement unit, but the operations listed in DOL. C_d^x is then defined as:

$$C_d^x = \sum_{o \in DOL} n_o * T_o$$

In each T_o the communication costs are included.

It is not possible to predict the absolut time savings for orthogonal migration because this depends on the degree cf parallelism within a system. In the worst case there is no other process that can be set up, in the best case time saving is equivalent to the offloading of the main processor by the migration candidates or even better if the execution on the coprocessor is faster than the original version. Therefore the cost functions given above can only be used to sort the objects of equal type (according to definition 3.6).

5 Conclusions

In this paper we have presented a system model that allows to describe the structure of systems to which vertical and orthogonal migration techniques are applied. In terms of this model we have discussed structural as well as quantitative aspects of both kinds of migration. This demonstrates the suitability of the choosen approach to serve as a base even for a formalization of the whole migration process.

6 Literature

[AND 81] Andrews, G.R.
Synchronizing Resources
ACM Transactions on Programming Languages and Systems, vol 3, no 4, October 1981, pp 405-430

[DAV 83] David, G., Graetsch, W.
A Hierarchical System Model for Vertical Migration
Submitted to IFIP Working Conference on System Description Methodologies Kecskemet (Hungary), May 23-27, 1983

[DIJ 75] Dijkstra, E.W.
Guarded commands, nondeterminacy and formal derivation of programs.
CACM 18, 8 (August 1975), pp 453-457

[HEN 81] Henry, S., Kafura, D.
Software Structure Metrics Based on Information Flow
IEEE Transactions on Software Engineering, vol SE.-7, no 5, September 1981, pp 510-518

[HOL 82] Holtkamp, B., Kaestner, H.
A Firmware Monitor to Support Vertical Migration Decisions in the UNIX Operating System
Proc. 15th Annual Workshop on Microprogramming, SIGMICRO Newsletter, vol 13, no 4, December 1982, pp 153-162

[JAM 77] Jammel, A.J., Stiegler, H.G.
Managers versus monitors
In: Information Processing 77, B. Gilchrist (Ed.). Elsevier North-Holland, New York, 1977, pp 827-830

[JON 78] Jones, A.
The Object Model: A Conceptual Tool for Structuring Software
In: Bayer, R., et al. (eds.): Operating Systems - An Advanced Course, Lecture Notes in Computer Science 60, Springer Verlag, 1978

[LAU 79] Lauer, H.C., Needham, R.M.
On the Duality of Operating System Structures
ACM Operating Systems Review 13 (2), March 1979

[LED 81] Ledgard, H.
ADA - An Introduction
ADA Reference Manual (July 1980)
Springer-Verlag, New York, Heidelberg, Berlin, 1981

[LUQ 80] Luque, E., Ripoll, A., Ruz, J.J.
Dynamic Microprogramming in Computer Architecture Redefinition
Euromicro Journal, no 6 (1980), pp 98-103

[PRY 82] Prycker de, M.
A Performance Analysis of the Implementation of Addressing Methods in Block-Structured Languages
IEEE Transactions on Computer, vol C-31, no 2, February 1982, pp 155-163

[STA 81] Stankovic, J.A.
The Types and Interactions of Vertical Migration of Funktions in a Multi-Level Interpretive System
IEEE Transactions on Computers, C-30(7), July 1981

[STO 78] Stockenberg, J.
Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/Software Systems
Computer, vol 11, no 5, pp 35-50, 1978

[WIR 80] Wirth, N.
MODULA-2
ETH Zuerich, Reports of the Institute for Informatics No. 36, March 1980

TOWARDS AN INFORMATION SYSTEM DEVELOPMENT ENVIRONMENT

Jan Dietz
Eindhoven University of Technology,
Department of Industrial Engineering,
P.O. box 513
5600 MB Eindhoven, the Netherlands

SUMMARY

The aim of IS development is to produce effective and efficient IS in an effective and efficient way. This paper deals with several aspects of effectiveness and efficiency, especially their establishment for evolving IS.
The need for appropriate intellectual aids and matching practical tools, together called the IS development environment, is being argued.
The use of prototyping, simulation and specification languages is emphasized, and there is some elaboration on the subject of specification languages.

1. INTRODUCTION

This paper describes ideas about IS and IS development, which the author considers useful to explore further. They constitute the framework for his research on IS development aids.
The aim of IS development is to produce systems that are effective, i.e. they behave as needed, and that are efficient, i.e. they operate without waste of resources.
The first research goal is to investigate the possibilities of specifying the behavior of a system such that the following requirements are met:

- **completeness**, i.e. a design, specified in this manner should contain all information needed for the subsequent detailing and realization;
- **consistency**, i.e. there shall be no conflict between parts of the description;
- **clearness**: there shall be no ambiguities; a point of special care will be the precise definition of the system's semantics;
- **formality**: a specification should be formally testable on completeness and consistency.

If one could specify a system in this manner it seems fairly feasible to generate its realization. Another motive for the research is that in the future not only people might use IS, but automated IS might use each other as well. This implies that a formal and precise specification of the system exists and is part of the system itself.

2. INFORMATION

Information systems produce information. Before discussing IS properties it seems wise to take up the subject of information first, because eventually the production of useful information is what it is all about.
A sound basis for studying the concept of information is provided by the science of semiotics (see e.g. [Morris 55] and [Nauta 72]). The central concept in semiotics is semiosis, this is a process in which something is a sign to some organism. The sign stands for something else (the referent) and causes effect in the agent of the process (the interpreter). Semiosis thus is a 'mediated-taking-account-of'.
The study of signs is subdivided into three fields:

- **pragmatics**: deals with the origin, uses and effects of signs;
- **semantics**: deals with the signification of signs;
- **syntactics**: deals with the form of signs without regard to their specific significations or their relation to the behavior in which they occur.

The notion of information is usually considered to be the effect a sign causes in the interpreter: if the sign has no effect, it doesn't contain information, if it has a great effect it is said to have a high informational value.

An essential condition for a sign to cause any effect is that its signification is understood by the interpreter.
In the same manner as the effect of a sign is conditionally determined by its signification, is its signification conditionally determined by its form, since signs can only be discriminated by their form. So pragmatic meaning presupposes semantic meaning and semantic meaning presupposes syntactic meaning. This is a very important point, since it shows that semantic meaning is carried by the sign's form and can be derived from its form only.
Semantic meaning also is something that the communicating interpreters must agree upon: signs do not possess any inherent semantic meaning. The major concern of automatic sign (=data) processing must therefore be to preserve semantic correctness and clearness.

In the remaining part of the paper the word 'information' is used to emphasize the pragmatic aspect of signs. The word 'data' will be used as a neutral term for signs.

3. INFORMATION SYSTEMS

Let us now focus our attention on the IS and its environment, and consider a situation like the one pictured in figure 3.1., consisting of:

- an object system (OS) as an organized part of the real world, in which activities are performed by agents (these may be human beings but also artifacts);
- an information system (IS), as the informational aspect system of the OS, i.e. the whole of operations, means and material aimed at the production of information to be used by the agents;
- an interface through which observations about the OS status and status changes flow into the IS and useful information flows from the IS into the OS. Next to this (functional) interface one may perceive an operational interface, through which the interaction between the agents and the IS takes place.

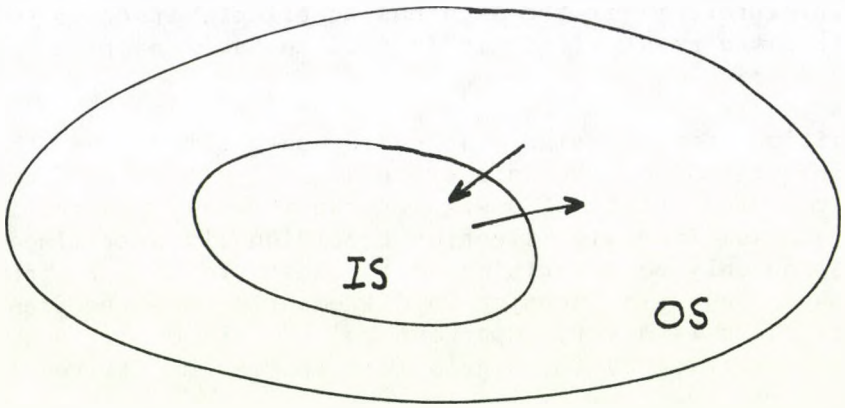


fig 3.1.

The boundary of the OS is considered to be wider than usually, and therefore needs some explanation. In fact it includes all agents to which data are sent by the IS and all sources from which data are received by the IS. When talking e.g. about an order system it includes the customers and the suppliers, when talking about a payroll system it includes the employees, when talking about a bank accounts system it includes the accountholders (customers) etc..

With the knowledge from the previous paragraph one could call an IS a sign processing and sign producing system.

If we consider the IS from the viewpoint of IS development we can make the observation that the IS and the OS must 'fit' together and that there will be something like a 'best' IS for a particular OS. What then determines the quality or fittedness of an IS?
It will be clear that a key factor must be the informational value of the signs which I consider to be determined by three factors:

- the effect of the sign, based on its signification: it must reduce the agent's uncertainty about what decision or action to take;
- the moment of receiving the sign: if it is too late it is of no use, the agent already had to take action;
- the agent that receives the sign: the sign should be sent to the agent that needs the information, and perhaps even is not allowed to be received by other agents.

Shortly one could say that the IS should produce the right sign on the right moment and deliver it to the right agent. The measure by which this goal is achieved is called the **effectiveness** of the IS.
The other, complementary, quality factor (called **efficiency**) is determined by the measure of consumption of resources like data, people, energy, storage capacity, processing power.
Many quality terms in vogue nowadays, like maintainebility, flexibility, userfriendliness, robustness etc. may more precisely be defined in terms of the effectiveness and efficiency factors.

In describing IS behavior I make a distinction between **functional behavior** and **operational behavior.**
The specification of the functional behavior includes the description of the output data, the input data, the stored data and the relationships between them.
The specification of the operational behavior basically states when output data are produced and input data are accepted.

Time plays an important role in IS, a role which seems often to be neglected. An IS contains a data model of the OS. There is however a delay in the accuracy of the model: a status change of the OS at time t1 is recorded in the IS at time t2 (t2>t1). For a semantically correct specification of the functional behavior of the IS it is necessary to take account of this delay, which can be done in two different ways.
The first one is to make use of the specification of the operational behavior of the IS. In the operational model one can force a process not to be executed until certain conditions are met. This approach is taken e.g. by PSL/PSA [Teichroew 77].
The other way is to record explicitly the origination time of data, an approach taken e.g. by DADES [Olivé 82]. I have a preference for the latter one because it is more rigorous.

4. DEVELOPING INFORMATION SYSTEMS

Effectiveness and efficiency change during the system's life. In fact they are always decreasing. The evolution of the OS as well as the emergence of new technologies lower the actual effectiveness and efficiency of the IS. As a matter of fact, but often overlooked: the very implementation of a new IS changes the OS (by definition!), evoking new needs and new possibilities, thus lowering its effectiveness.
Developing IS therefore is an endless process: there always is a 'solution' and there always can be found a better one.

In contradiction with many other authors I can, when talking about systems development, only distinguish between two essentially different activities: **design** and **construction** (=realization).
The design process can best be defined as the creative activity of concurrently studying problems and generating solutions [Alexander 70]. By problem is meant any situation in which there is perceived to be a mismatch between what is and what might or could or should be. The design process shows a constant alternation of analysis and synthesis, intertwined and distinguishable but not separable.
The point that I would like to stress is that there cannot exist requirements or needs distinct from solutions or fulfillments, because a requirement or need can only be expressed in terms of solutions: both requirements specifications and program specifications are design specifications, they only differ in the level of detail. This may sound embarrassing to people, who like to consider the expression of the problem and the specification of the solution as two really separable activities.

During the design process the designer constantly takes design decisions, each design decision being a step forwards to the end solution. At the same time the problem gets better defined and the set of possible solutions is reduced. Initially one starts with an empty problem and thus an infinite solution space. However from the very first contact of the designer with the problem area, the problem gets shaped and big parts of the solution space are cut off.
At every step the designer should strive for a minimal reduction of the solution space. This needs creativity and a permanent resistance to time pressures and the habit of following known patterns. However proceeding in this way is a prerequisite for achieving 'quality' systems.

Developing IS also rarely is developing from scratch. Nearly always it will be a matter of modifying and extending the existing 'solution'. This stresses the point of precise specifications of a system's behavior and thus the need for specification languages.

5. INTELLECTUAL AIDS TO IS DEVELOPMENT

Developing IS is dealing with multitude and complexity, which makes it necessary to expose the problem situation from several different viewpoints, an approach advocated e.g. by Ross [Ross 77]. There are several intellectual aids well identified now for dealing with multitude and complexity. In [Krakowiak 78] they are listed for the area of program development:

- **decomposition** of a complex object into more manageable parts is an old methodological principle. However it must be conducted in a systematic fashion and appropriate guidelines are needed;
- **abstraction** is the intellectual operation whereby a representation, or abstract model, of the behavior of a complex object is constructed, which only retains some relevant properties and omits irrelevant ones;
- **refinement** is the process by which abstract objects are eventually implemented. The elementary refinement step is to construct an object in terms of more primitive objects by the application of a set of composition rules.

Next to these general aids I find two ideas particularly appealing and useful for the area of IS development.
One of them is the level model [e.g. Berg 79], which is elaborated into the **engineering paradigm** by [Ramamoorthy 78]. The creation of a solution to a problem is viewed as a transformation P(needs)=product. In a large and complex design situation, different phases are gone through and the transformation takes on a number of distinctly recognizable forms:

```
needs = Form0
P1(Form0) = Form1
P2(Form1) = Form2
        .
        .
        .
Pn(Formn-1) = Formn
Formn = product
```

The paradigm shows the systematic evolution from the first, coarse, design (needs) to the last, fine, design (product). The idea incorporates decomposition, abstraction and refinement, and it would particularly be useful if each Formi can be expressed formally and if each transformation can be verified formally.

The other attracting idea is that of viewing an IS in each of three different domains [Winograd 79]: **subject domain**, domain of **interaction**, and **domain** of **implementation**. Each viewpoint is appropriate (and necessary) for understanding some aspects of the system and inappropriate for others.
In the subject domain the universe of discourse is described: the objects and processes in the OS of which the IS is to be a model. In this domain the functional behavior of the IS is defined.
In the domain of interaction the relevant objects are those that take part in the system's interaction with its environment: users, files, questions, answers, forms etc.. The processes to be described are those like querying the system and performing a system function. In this domain the operational behavior of the system is specified.
The behavior within the boundary of the IS is seen in the domain of implementation. A description in this domain consists of specifications of (sub-) components and the interactions between them. This, I think, can recursively be considered a system, that can be viewed in each of the three domains.

6. PRACTICAL TOOLS NEEDED

The professional system designer clearly needs help to perform his task, a help which can be provided by a set of well chosen intellectual aids, supported by a set of matching tools, together called a **system development environment**.
This environment must be helpful in establishing and maintaining effective and efficient IS and must support all distinct design and construction phases. There are three topics that deserve special attention.

The first one is what I would like to call the **functional quality** of an IS. It means that the IS produces the right information and that it makes to that end efficient use of the data resources (input data and stored data). A very powerful tool to support the activities concerning the establishment of functional quality is the technique of (rapid) **prototyping**.

The second topic, strongly related to the previous one, is what I would like to call the **operational quality** of an IS. The concern in this aspect is that the information is produced at the right moment and delivered to the right agents, and that all input is processed in due time. The main variables in an operational model are the processing and storing capacities of the physical resources used. A well-known and powerful aid for this kind of work is **simulation** [e.g. Bodart 79].

The remaining topic is that of the preservation of the IS semantics during the subsequent design steps up to the final, realizable, one. More generally stated it is the problem of establishing the **semantic equivalence** of two different specifications of the same system.
Conforming to the engineering paradigm one would need several specification languages, each of them best fitted to a certain level. The particular concepts and constructs wanted in a specification language depend heavily on the specific application area. To meet the need for variety in specification languages, I prefer to think of either universally applicable formalisms, containing a very limited set of primitive concepts and constructs and the possibility to define new ones (e.g. SDLA [Knuth 82], or a meta system for the generation of arbitrary formalisms, like SEM [Teichroew 79]. I think that both approaches offer a basis for a rigorous definition of the semantics of specification languages.

The problem of identifying a particular level (for a particular application area) and the corresponding specification language is equal to the problem of determining the right level of abstraction. It will be clear that in my view satisfactory solutions can only be given by the 'best' designers.
I see many formalisms or models in use or proposed lacking the right level of abstraction. Let me take the ERA-model (Entity-Relationship-Attribute) as an example to illustrate what I mean. The ERA-model leads, as I see it, to a premature and often unconscious decision about how values of object properties are stored and thus to an unnecessary limitation of the design freedom in the steps to come.
Attribute values in the ERA-model are tought of as record elements, more strongly connected to the object than relationship values, which are mostly seen as separately stored data.
A right level of abstraction in the early design steps would be to consider all property values as function values, e.g. articlename = f1(article), but also: averagestock = f2(article,period). In this way one abstracts from how the values are produced. The idea of access-functions is very well described in [Abrial 74].

REFERENCES

Abrial 74 — J.R. Abrial: 'Data Semantics', in: Data Base Management, W. Klimbie and K.L. Koffeman eds. North-Holland publ. (1974)

Alexander 70 — C. Alexander: Notes on the synthesis of form. Harvard University Press (1970)

Berg 79 — H.K. Berg: 'Towards a uniform design methodology for software, firmware and hardware', in: The use of formal specification of software, W. Brauer ed. Springer Verlag (june 1979)

Bodart 79 — F. Bodart, Y. Pigneur: 'A model and a language for functional specification and evaluation of information systems design', in: Proc. IFIP TC8 WC on formal models and practical tools for Information Systems design, H.J. Schneider ed. North-Holland publ. (april 1979)

Knuth 82 — E. Knuth, F. Halász, P. Radó: 'SDLA, system descriptor and logical analyzer', in: Proc. IFIP TC8 WC on comparative review of information systems design methodologies, T.W. Olle, H.G. Sol, A.A. Verrijn-Stuart eds. North-Holland publ. (1982)

Krakowiak 78 — S. Krakowiak: 'Methods and tools for information systems design', in: Information Systems Methodology. Springer Verlag (1978)

Morris 55 — C. Morris: Signs, language and behavior. G. Braziller, New York (1955)

Nauta 72 — D. Nauta: The meaning of information. Mouton, The Hague/Paris (1972)

Olivé 82 — A. Olivé: 'DADES, a methodology for specification and design of information systems', in: Proc IFIP TC8 WC on comparitive review of information systems design methodologies, T.W. Olle, H.G. Sol, A.A. Verrijn-Stuart eds. North-Holland publ. (1982)

Ramamoorthy 78 — C.V. Ramamoorthy, H.H. So: 'Software requirements and specifications, status and perspectives, in: Tutorial on software methodology. IEEE (1978)

Ross 77 — D.T. Ross, K.E. Schoman: 'Structured analysis for requirements definition', in: IEEE Trans. on S.E. vol 3,1 (jan 1977)

Teichroew 77 — D. Teichroew, E.A. Hershey III: 'PSL/PSA, a computer-aided technique for structured documentation and analysis of information processing systems', in: IEEE Trans. on S.E. vol 3,1 (jan 1977)

Teichroew 79 D. Teichroew, P. Macasovic, E.A. Hershey III, Y. Yamamoto: 'Application of the entity-relationship approach to information processing systems modelling', in: Entity-Relationship approach to systems analysis and design, P.P. Chen ed. North-Holland publ. (1979)

Winograd 79 T. Winograd: 'Beyond programming languages', in: Comm. of the ACM vol 22,7 (july 1979)

CONCRETE USE OF ABSTRACT DEVELOPMENT FORMALISMS

R.E.A. Mason

Department of Computing and Information Science
University of Guelph
Guelph, Ontario, Canada, N1G 2W1

April 15, 1983.

ABSTRACT

The literature of Software Engineering demonstrates a wide variety of approaches to systems development amongst scientists and practitioners who cannot communicate effectively amongst themselves. This paper discusses the need for agreement on a taxonomy of programming, as an aid to better communication. Using a taxonomy as a framework for discussion, the paper reviews some of the the current ideas on formalism, and proposes that methods currently in use, especially in the development of Interactive Information Systems, represent valid abstract formalisms which can contribute ideas of value to other domains of software engineering.

1. INTRODUCTION

The Proceedings of the 6th International Conference on Software Engineering contains a great many papers of interest to practioners and software engineering scientists. It also illustrates, like much of the recent literature, a serious problem in software engineering: namely that it is very difficult to understand whether progress is actually being made in this field. In his classic Turing Award lecture ten years ago (1), E.W. Dijkstra described as a very real possibility, his vision that "well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining programming ability, at the expense of only a few percent in man-years of what they cost us now, and that besides, the systems will be virtually free of bugs". In his lecture, Dijkstra demonstrated that the problems were generally recognized, and that there was an economic need for solution of the problems. In his lecture he presented arguments in support of the technical feasibility of solutions to the programming crisis.

It is self-evident that this vision of the future has not come to pass, at least for many who are still proposing tools, techniques, ideas which strive towards cost-effective production of better computer programs. Consider the probable reaction of a "typical" programmer in the data-processing department of a Canadian insurance company to some of the papers presented at the 6th International Conference on Software Engineering:

* Greenspan, Mylopoulos and Borgida (2) "adopt the view that software requirements involve modeling of considerable real-world knowledge, not just functional specifications." They propose a framework which allows information about the real-world to be consistently recorded

and manipulated to describe an applicaton. Of course, our programmer knows this from empirical observation. He notes that the authors intend to set up a structured lexicon of terms relevant to the domain of discourse (a hospital). He notes they have other research underway on aspects he knows need solution. He wonders whether it will be successful; and if it will ever apply to his application domain.

* Boehm, Elwell, Pyster, Stuckle and Williams (3) present an overview of the TRW integrated software support environment, a range of TRW tools, and the study which resulted in development of this sytem. Our programmer applauds the comprehensiveness of the approach, which projects a four times improvement in programmer productivity by 1990. Perhaps his company should get TRW to develop its next system?

* Bauer (4) advocates strict formalization in the program construction process, based upon one specification, which will be transformed into a correct program. Our programmer notes that Bauer understands the problem well, gives proper emphasis to the need for the client to understand the specification, etc. But he wonders whether his boss will go to court to have it proven that the client did indeed agree to the specification, or whether both he and his boss will simply be fired by the insurance company, when it turns out the client did not get "what he wanted."

Are these papers relevant? Are the many others (a few as good as those mentioned here) in the 6th International Conference on Software Engineering relevant? Are the very many other software engineering papers published each year relevant to our programmer? I do not answer these questions, because our programmer is hypothetical. What does matter is that, while the authors of the papers appear to be in agreement on some matters of importance, it is difficult to know the bounds on that agreement. Let us consider posing, to the authors of each paper the question: would you propose to apply your ideas to a typical development in a typical Canadian insurance company, within the next 5 years?

2. TAXONOMY OF PROGRAMMING

The authors of the papers cited cannot, I believe, answer such a question. At an ACM Workshop on Rapid Prototypng (5) in 1982 a half-day was spent by the participants discussing in detail, and with vigour, variants of the "waterfall" system development cycle. There was heat, but little light. Experts disagree what the stages of development are, let alone what they ought to be. A longer period was spent discussing what a "prototype" is; although many of us have written papers on the subject, we cannot yet be certain there is a subject. We would certainly, therefore, agree that the hypothetical programmer introduced above is sufficiently ill-defined that the question cannot be answered! Yet, one has a vague feeling that all the last three authors cited, and perhaps even the first, intend their work to be relevant to the "typical" data-processing context. Someday, if not now.

I propose that it is desirable, perhaps necessary, to establish some classification of characteristics important to software systems development, so that those interested in the subject can understand one another; so they

can understand each others' assumptions; so they can communicate. Others have proposed classifications (6,7) which serve valid but limited purposes. The purpose of the classification I suggest below is to understand what programming is.

Definitions of programming are personal and determined by the individual's goals. One programmer has finished his work when an algorithm is written (correctly) on paper. Another must produce a system which will deliver reports to a client. A third must alter an existing system to meet a new requirement. What is needed is a classification which will permit precise, (or more precise) discussion of such distinctions. We require the ability to communicate precisely about our shared concepts. We require a language, one which assigns special meanings to common words.

2.1 Dimensions of a Programming Taxonomy

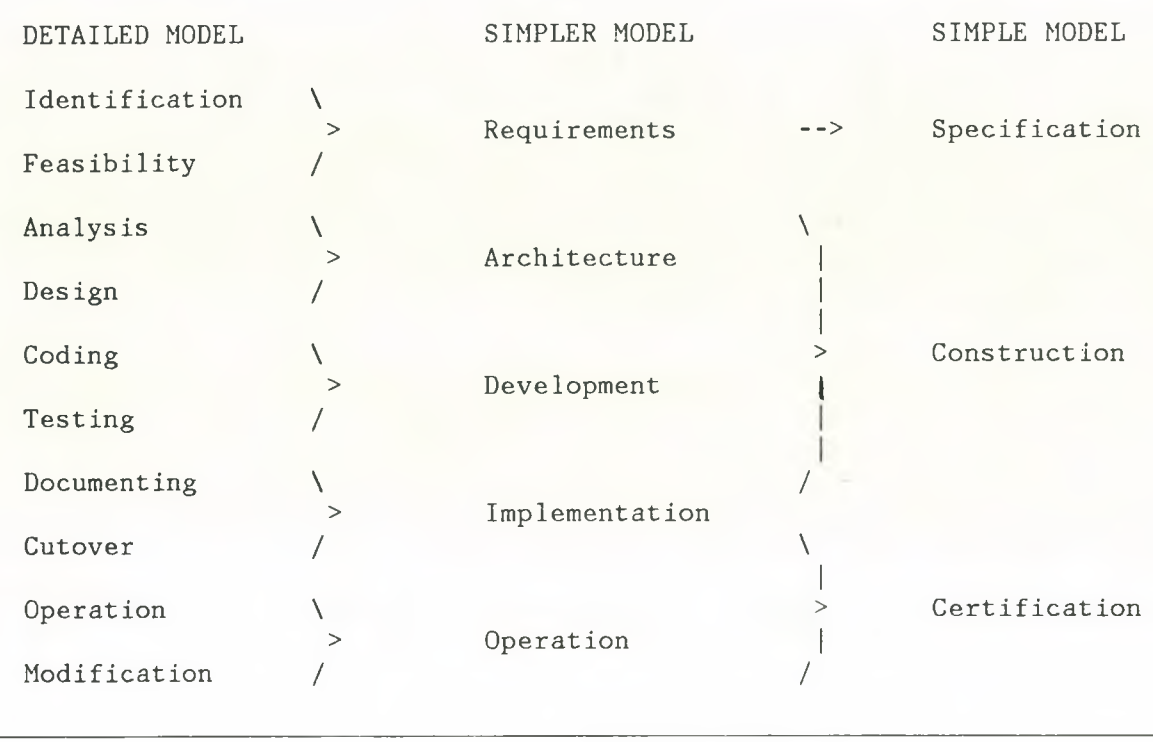
The theory and practice of programming may be viewed from many perspectives. What follows is an attempt to describe three such perspectives or dimensions, and then to define in more detail each of those dimensions.

The dimensions delineate separate sets of concerns. The sets chosen are felt to encompass important aspects of programming, though perhaps not all such aspects. The dimensions considered are the program development cycle, programming domain, and programming resources.

2.1.1 Program Development Cycle

It is generally agreed that it is important to consider the stage of development of a program; the terms used to describe the specific stages, i.e. the model for the cycle itself, has no general agreement. Figure 1, below, suggests three such models all of which are familiar, and possibly acceptable. Other models have been proposed (e.g. 8), for programming domains which are not so often discussed.

Figure 1. Three Models of the Program Development Cycle



It is clear that, in many programming domains, such models may be mapped to one another. When one comes to consider less traditional models, say one involving prototypes, the agreement amongst authors on the description of the Development Cycle dimension often breaks down. Note, in the models presented here, the "Operation" phase is ambiguous: another kind of problem. This paper does not discuss the Program Development Cycle dimension of a taxonomy further.

2.1.2 Programming Domains

The object of the Domain dimension is to permit differentiation amongst a reasonable number of kinds of programming problems. There are a variety of problem domains within which the development of computer applications takes place, and the degree of success achieved with a specific programming methodology seems to depend upon the domain within which it is applied. This second dimension of a programming taxonomy, characterizing "typical" domains, is suggested in order that discussion of such dependencies may be orderly. The names here attached to each domain are personal to this author, but the descriptions may be more definitive. Five such Programming Domains are discussed below, each representing a blend of problem size and problem type.

(i) Complex Applications

This domain is concerned with functionally complex computer applications, usually on a very large scale. The application may be highly specialized, such as an airline reservation system. Or it may be multi-faceted, as in an automated military defense system. The important characteristic is a great complexity arising from organizational, geographical, or technical application factors. Accompanying this complexity degree of specialization among personnel involved in development. The development system described by Boehm (3) seems to be directed towards a Complex Application domain.

(ii) Data Processing

This domain includes operational automation computer applications in business and government. Its characteristics include an application emphasis on data manipulation, numerous interfaces with existing systems, and problems relating to the understanding of client requirements. Modification and evolution of existing operational programs is often a critical concern.

(iii) Information Systems

This domain is intended to characterize situations in which the major programming problems concern data access and presentation, rather than data manipulation. Interactive Information Systems (IIS), as defined by Wasserman (6) are one example. Applications such as the development of a microcomputer spreadsheet package might be another. One would expect applications where presentation concerns predominate to be developed using different tools, languages, development cycles, etc.

(iv) Scientific Programming

This domain is concerned with providing computer facilities to aid research, analysis or experimentation in any field of endeavor. Whereas the Data Processing and Complex Application domains are concerned with providing stability of functional service (with some slow evolution), the scientific domain is most often concerned to provide rapid evolution. Indeed, "Dynamic Programming" would be a better label, but this phrase has another meaning. In this domain, the solution of some problems typically requires the discovery or selection of an appropriate algorithm. "Problem Solving" or "Decision Support" applications in business are also included in this domain.

It is unusual to find large groups of programmers working on a single problem of this type, but there may be individual workers in many different locations all working on the same problem. This aspect results in great value being attached to correctness, comprehensibility and other program characteristics which are often not overriding in other domains.

(v) Individual Support

It is useful to complete the spectrum of domains by envisaging a one man/one computer situation in which the computer is used to provide personal support. This, then, is the natural environment of the personal microcomputer system.

A microcomputer-like Individual Support domain is also the target of many data

processing application subsystems which attempt to provide programming facilities (query languages, report generators, etc.) for non-data processing "users". The key to this domain is a set of facilities which are easy and natural for people to use. The emphasis is on supporting programming by people not trained in computing.

In summary, the five domains described above represent a spectrum of situations within which programming occurs. The first four domains often employ formally trained "programmers" and "analysts", but the expectations from such persons are different in each domain. Not many real-life situations will cleanly fit within a single such domain, yet most real-life situations can be identified with these categories. This paper returns to a discussion of the programming domains, but a third dimension of the taxonomy is introduced first.

2.1.3 Programming Resources

Adopting a broad definition of programming, the programmer is any person engaged in building systems. In the domain of Complex Applications, individual programmers will provide highly specialized skills with limited personal scope of influence over the entire system. Some programmers will design, some will code, some will test, others will manage. All will be specialists. At the other extreme, in Individual Support domains, one person performs all these functions, and is also user. The third dimension of the programming taxonomy includes the various resources, human and material, which are brought to bear on system building in all domains.

The primary resources involved in programming, within all programming domains, involve skills possessed by individuals, and facilities to enhance these skills. Because computer programming is labour intensive, it is useful to distinguish a few different "programming" skills which are required, in addition to the other types of resource. Specific resources which are commonly needed in all domains include the following:

* Problem Skill, which relates to the understanding of the nature of the problem and application of analytical skills to the solution of the problem.

* Programming Skill, which is the ability to effectively use design techniques and programming languages. Programming Skill is a requisite for all programming environments. The objective of the Individual Support domain is to eliminate the need for formal training to develop this skill.

* Communication Skill, which is the ability of the programmer to communicate with colleagues working on the same problem.

* Management Skill or resource is the ability to organize and direct the application of the other resources.

* Application Structure is perhaps the least recognized important resource available to programmers. It is the key to design of programs in both the Data Processing and Scientific Programming domains. Application structure is necessary to the mangement as well as design of Complex Applications. As is suggested below, effective methodologies may recognize and take

special advantage of this resource.

* Methodologies and Tools represent other resources which, like Management Skill, achieve their effects through their influence on the use of other resources. Computer cycles, storage space, language processors, and conceptual approaches all fall into this category.

* Time is the one depletable resource in any system development. The Programmer and Management resources (skills) may indeed be enhanced by the passage of time: time permits experience to be gained.

2.2 Why These Dimensions?

Other categories of factors which influence programming success are certainly possible; the taxonomy presented here is, as previously noted, intended to be illustrative. But it is not completely arbitrary. The attempt should be to agree upon some taxonomy which exposes issues of importance in software engineering. A large concern of that field is to identify tools, or systems of tools, or methodologies, or management techniques, which will improve programming effectiveness. If the taxonomy can be shown to be useful in describing why some tools work, and if it can help identify new tools which prove to work, the taxonomy can represent a basis for a valid theory.

3. METAPHYSICAL (ABSTRACT) FORMALISMS

Bauer, in the paper previously cited, has argued that the need for formalization is inherent in computer science, and arises because "the computer itself is absolutely formal in its contact with people." Bauer develops the content of the three-stage development cycle which is called, in Fig. 1 above, the "Simple Model." He then proposes that "the formal essence of rational specifications" can alternatively be expressed by algebraic abstract types, by predicate logic, by non-determinism, or by higher order functionals. He illustrates use of the first of the three tools, but is forced, in his discussion of the consequent stage three, Certification, to introduce discussion of the role of "Judge", a legalistic approach which will not be encouraging to those attempting to have programs developed for them.

Bauer's approach, as it regards Programming Domain, seems primarily from the perspective of what is above termed Scientific Programming. Despite allusion to other programming domains the assumptions of his methodology make this probable. (Absence of formal discussion of programming domains in much of the published work often forces us to infer the domain of application.) Furthermore, Bauer's discussion of the resources used for the programming task is also informal, and the absence of a structured view leads to some confusion. Bauer notes that "high-ranging people (lack time to) acquire the fundamental knowledge of a computer scientist." It appears also that computer scientists lack the will to acquire knowledge of alternative formalisms. The resource dimension of the programming taxonomy attempts to separate the various skill concerns in program development, recognizing that these different resources will be supplied from different sources depending upon both the Programming Domain, and the stage of the Development Cycle. The next section of this paper presents an alternative

formalism which has proven to be of great value in actual Information Systems domains. Section 5 compares some aspects of this formalism and that of Bauer, in the framework of the taxonomy.

4. FORMAL DEVELOPMENT OF INFORMATION SYSTEMS

The methodology presented below is in fairly wide use in industry for the development of IIS systems. Informal approaches which are highly similar have been employed for some time. The more formal approach, developed primarily by Art Benjamin of On-Line People, has been described previously by Mason and Carey (9) from which the description below has been adapted. A commercially available tool which supports the method is described in (10).

4.1 Architecture

The methodology is called Architecture-Based Methodology, and it takes its name from the analogy with the architected approach to a building or other structure, in the manner suggested by Ross and Schoman (11). The essence of the methodology is that the system designer, or architect, develops a view of the system based on its external description or appearance. The designer works inward from this view, to develop system details always consistent with the external appearance of the system. The important role of the system designer in the earliest stage of development is, like the role of the architect, finding a realistic expression of the system's appearance which is both understandable and acceptable to the users. Traditional methodologies tend to emphasize acceptability and function at the expense of understandability, but of course a system description which is accepted, but not understood, is not really accepted.

This methodology emphasizes, from the beginning of a project to its conclusion, the overriding importance of the user's ability to understand the developers' interpretation of his requirement. That is, there is a great emphasis on the Communication Resource of the taxonomy.

Thus the description of the external appearance must be embodied in some form of specification which is capable of complete and unambiguous interpretation by the users. Such a specification is analogous to the architect's drawings or scale model for a building: it is an effort to communicate to the users (or the customer) within a discipline which also provides consistent but more detailed descriptions for the engineers and builders who will later build the actual structure. An approach which employs an interactive screen-oriented scenario, which behaves like the proposed system, is a direct and simple solution to this problem for IIS projects.

4.2 Transaction Screen Perspective and Dialog-Based Design

The second element of the architecture approach is the adoption of a common view or design-concept for the underlying structure of all applicatons. As the building architect keeps in mind fixed concepts of how a house will be constructed, so the system architect has fixed views of appropriate structures which apply in well-understood situations. The transaction screen perspective, the view that the application consists of a series of

Input-Process-Output sequences of screens, is a key to effective development of many business systems, particularly IIS. The linkages between screens in a sequence may be data-dependent in some instances, and fixed in others, but these are elaborations. The operational user interacts with the system in a dialog, viewing sequences of screens, entering data into fields in screens, and being concerned only with the behaviour of the data and the screens.

The screen-oriented dialog perspective provides strong support to architecture approach, since it is easy to implement screens and sequences of screens, if the data content is fixed. The concept of the "scenario" begins here, as a special class of prototype in which a sequence of computer-display screens behaves exactly as the final screens in the application system are intended to behave. In the scenario, however, the user must follow a fixed script, since the application logic is not yet implemented.

Clearly, such scenarios are an improved means of communication with the users of a proposed system. Adoption of the transaction screen perspective, which permits easy development of such scenarios, represents an attempt to exploit the Application Structure resource inherent in the Information Systems Programming Domain. It also constrains the applicability of the methodology.

4.3 Project Management

In addition to adopting a conceptual approach (architecture) and a design approach (transaction oriented screen dialogs), a complete methodology should consider the project management and control approach within which the design will be articulated. That is, the Programming Resources should be formally considered by the methodology. The Architecture-Based methodology is directed to a management environment in which the user and developer roles are distinctly different. This is not a methodology intended for do-it-yourself programming by end-users. In fact the methodology is intended to support three separate and distinct roles.

The user role is to determine the functional needs of the system, and to understand completely the external appearance of the system. The user, who may be at times the operational user and at other times the user manager, concentrates on user concerns: function, operational sequences, timing and performance, usability, etc. Tools and methods used in determining the system specification are directed at ensuring the user's full and complete appreciation of these concerns.

The developer's role is to achieve an accurate, complete and timely translation of the system specification into a working product. The developer role is thus the traditional one for managed software development environments, although the nature of the work itself may be non-traditional because the tools used are new. For example, the system specification is represented by a series of machine-implemented application scenarios, rather than functional flow-charts or application structure diagrams. Using tools designed for this purpose and adopting the program structures just described, a good deal of traditional design and development may be eliminated.

The building of understanding between user and developer can be assisted by tools and methodology, but this understanding cannot be left to tools alone. A third and bridging role is essential to strengthen the Management Resource typically available within the IIS Programming Domain. This is the role of architect. The architect has responsibility for ensuring that the user(s) understand the system specification, and that the developer(s) deliver the product specified. This role is supported both by tools and by the personal characteristics of the architect. The architect must develop adequate trust among the three parties, and must maintain the integrity of the development process by adhering to the methodology.

4.4 Iterative Design

The final element of the methodology is the view it adopts of the Development Cycle. Development of interactive systems must be viewed as an iterative process. The user's understanding of requirements in the business environment normally evolves rapidly, especially where new approaches, such as IIS applications typify, are involved. A process which exposes the user to life-like scenarios of the final application will lead to wide exploration of application alternatives during the earliest stage of development. The development cycle will emphasize efforts during this requirements stage. Iteration at this stage, when supported by effective tools, will reduce later costs.

The benefits of an iterative approach are strongly supported in the literature relating to prototypes. The Architecture-Based methodology considers three specific levels of iteration within the specifications phase, and then proceeds to system development. The first set of iterations make use entirely of scenarios constructed from sequences of fixed-information display screens. Since the scenario screens must simulate user-computer dialogs, it is essential that these fixed-information screens be capable of accepting user data. However, no data analysis occurs on entry, and the scenario proceeds according to a script developed by the system architect. User and architect iterate on scenarios until an adequate first-level representation of the application is reached. This iteration process leads to agreement on such matters as screen-flow sequences, screen content, and whether the application is to be menu-driven or forms-driven, question and answer, etc. It also clarifies details including screen layouts. The user gains an excellent appreciation, where the architect is skilled, of the options available, and their implications for the application.

Many applications require that particular attention be paid to the details of data-dependent calculations. In these cases, a second-level iteration is required, in which actual database interactions and application computations on limited samples of data occur. This is considered to constitute a demo or demonstration, and assists in the clarification of both application logic and more detailed screen-flow sequences. The demo phase represents a partial implementation of the full application. Emphasis is on quick implementation of key or controversial areas of the system. The architect works with both users and developers during this stage.

Often, at this point, many relatively straightforward parts of an

application will not have been seen by the user in scenarios. That is, the specification will be incomplete in its details. A final series of iterations then may take place on a complete specification of the system. Application logic may be implemented; error-handling and recovery procedures are specified, and a "prototype" of the entire application is prepared. This prototype is exercised for users, and evolves to become the final system specification. The architecture-based methodology considers this prototype as being what Keen and Gambino (12) call the version 0 release of the system. Carey and Mason (13) have reviewed the meanings attached to the word "prototype" in some of the recent literature. They discuss in more depth distinctions between the various kinds of prototype referred to in the above description of this methodology.

5. METHODOLOGY AND FORMALISM

The IIS methodology has been demonstrated by considerable use to have great value within its intended Application Domain. What makes it formal? The language employed for specifications, namely a scenario of the desired product, appears to be effective, but informal. Most of us would agree that this specification language is far from being as formal as, say, the abstract algebraic specification of Bauer. This is not, however, the case. The dictionary meaning of "formal" refers to the essence of a thing. "Of the outward form, shape, appearance, arrangement, or external qualities..... explicit and definite." (14) What could be more explicit or definite, what could better describe the form and external qualities than a scenario?

The architecture methodology is as formal for the Information Systems domain, as is the abstract type for the Scientific domain. Both instances require that the user and developer agree upon a mutually-comprehensible specification language. The important characteristics of that language include precision and compactness, in addition to comprehensibility. The existence of tools supporting the specification language, as in the case of the IIS architecture methodology, enforces the formalism which users and developers must both agree upon.

Bauer notes that the formal specification is the "pivot". Construction of the actual program can indeed, with appropriate tools, be performed mechanically. In the IIS case, tools do exist to perform such translations; in the case of abstract algebraic types, mechanical translation capability exists, perhaps, if the algebraic (specification) language is appropriately selected.

5.1 Comparing Two Formalisms

If it can be agreed that the architecture approach is indeed sufficiently formal, it should be examined to determine whether it has anything to contribute to other programming domains. It is remarkable that the methodology for IIS proposes three development stages. These stages appear to map well to the three-stage process proposed by Bauer:

Bauer	Architecture
Specification	Scenario
Construction	Iteration on Demos
Certification	Version 0

Since the two approaches seem to describe similar approaches to similar problems, it is tempting to believe they are equally valid, each in its own Application Domain. This is not the case; I would propose that the architecture approach has a significant advantage. Each methodology embodies an analogy. In the architecture methodology this has been noted, and is imbedded in the name. Bauer's methodology might be termed "judicial". Both approaches recognize the great importance of a professional approach, but Bauer chooses a different profession.

The essence of the difference is that the architecture approach clearly identifies the need for independent professional guidance throughout the development process. In effect, this approach argues for the creation of an important new profession: that of programming architect.

If this idea is applied to Bauer's approach, it improves the liklihood of successful systems being developed. Rather than judging, upon conclusion of the product construction, whether the "contract" has been fulfilled by the computer professionals, the customer would hire a programming architect to ensure, throughout the specification and construction, that the specification is continuously adhered to. This is simply a restatement of the view that verification should be a continuous process which occurs throughout the development cycle. Client managers and computer scientists do agree on this. The formal architecture methodology suggests a means of achieving it.

It was suggested in 4.2 above that the inherent "structure" of a programming problem represents a valuable, and often overlooked, "resource" for the programmer. In the Scientific Programming domain, such structure is often directly reflected in use of a design language which is itself formally structured. In the Information Systems domain, similar structure can be imposed upon the problem, and results in the opportunity to achieve great benefits. The methodology for IIS presented above exploits such structure by adopting the Dialog-Based Design structural model. In effect, some of the "how" was fixed prior to deciding "what". While mathematical expression of its formality has not been developed, there can be little doubt that both the nature of the Application Structure resource and the benefits of exploiting it are similar to what is achieved by mathematical approaches in Scientific Programming.

Finally, it was noted above that the scenario as specification may be incomplete. This may dissatify those who have considered that completeness is a critical specification attribute; however such considerations are highly Domain-dependant. There is a natural conflict between completeness and comprehensibility in specifications, and the IIS (and possibly the Data Processing) domains demand more of the latter than of the former. If user and developer have agreed upon methodology, formalism is not sacrificed.

6. DISCUSSION

This paper has proposed a taxonomy of computer programming, and demonstrated how it might apply to discussion of two quite different programming methodology proposals. The comparative discussion was aided by the fact that both methodologies employ three-stage Development Cycles which are obviously similar in intent. The domain of programming concern of the two methodologies is quite different, as are the Programming Resources emphasized in each case.

The idea is intriguing that a methodology intended for use in a Programming Domain having "appearance" rather than "substance" as a major concern might offer something to to other domains. At the ACM SIGSOFT Workshop on Rapid Prototyping it was evident that workers in the IIS field had achieved a more advanced degree of implementation of formal approaches; the one described here is but one example. A major argument of this paper is that these approaches may indeed be formal, where they are based upon agreed and enforced use of a particular specification language. Mathematical formalisms are not appropriate in some Application Domains, and they are certainly not the only valid formalisms.

The taxonomy is an attempt to permit analysis of the different characteristics which specific programming projects may have, in a manner which can lead to valid conclusions about programming in the general case. Although use of the taxonomy as a framework for comparison of approaches can be fruitful, it is premature to claim that the taxonomy can form the basis for a theory of programming. Indeed, it is far from certain that the dimensions of the taxonomy are even the appropriate ones. Nevertheless, some such approach may be promising.

The problems with any conceptual subdivision of a complex activity into components are many. One set of problems arises from the words which we use to label concepts. It is very difficult to enforce the discipline implied by new definitions for old words, except in very narrow fields. Programming is not a narrow field. A second kind of problem arises from the lack of shared experience. Workers in one Application Domain rarely possess the direct experience of another domain which permits effective communication in the absence of agreement on terminology.

A third problem in taxonomy development is sociological. There is little incentive for the documentation of formal concepts relating to domains which may appear to some to be undisciplined and chaotic. The effort to understand and describe these formalisms is great, since they are not founded on two-dimensional mathematical logic. The rewards in this area are not in description on paper of formalisms ill suited to such a medium. However, the economic and societal importance of understanding (and correcting) the problems of such domains is great. A purpose of this paper is to illustrate that this task is not only possible but that it may contribute to progress in more structured domains.

Concrete uses of development formalisms should be sought out and studied. They have much to contribute to computer science.

7. ACKNOWLEDGEMENT

This work has been supported by the Natural Sciences and Engineering Research Council of Canada, under grants A3045 and A5547.

REFERENCES

(1) E.W. Dijkstra, "The Humble Programmer", CACM 15 (October 1972), pp. 859-866.

(2) Greenspan, S.J. and J. Mylopoulos, A. Borgida, "Capturing More World Knowledge in the Requirements Specification. Proc. 6th International Conference on Software Engineering. Tokyo September 1982, pp. 225-234.

(3) Boehm, B.W., J.F. Elwell, A.B. Pyster, E.D. Stuckle, R.D. Williams, "The TRW Software Productivity System". Proc. 6th International Conference on Software Engineering. Tokyo September 1982, pp. 148-156.

(4) Bauer, F.L. "From Specifications to Machine Code: Program Construction through Formal Reasoning." Proc. 6th International Conference on Software Engineering. Tokyo September 1982, pp. 84-91.

(5) Zilkowitz, W.V. ed., "Workshop Notes, ACM SIGSOFT". Workshop on Rapid Prototyping. Columbia, Maryland, April 19-21, 1982.

(6) Anon, "Quantitative Software Models", Data Analysis Center for Software, Rome Air Development Center, March 1979.

(7) Houghton, Jr., R.C. "Software Development Tools", National Bureau of Standards Special Publication 500-88, February 1981.

(8) Brittan, J.N.G., "Design for a Changing Environment", The Computer Journal, Vol. 23, No. 1, January 1979, pp. 13-19.

(9) Mason, R.E.A. and Carey, T.T., "An Approach to Prototyping Interactive Information Systems", Proc. 3rd International Conference on Information Systems, Ann Arbor, Michigan, December 1982.

(10) Mason, R.E.A., T.T. Carey and A. Benjamin, "ACT/1: A Tool for Information Systems Prototyping", ACM Workshop on Rapid Prototyping, Columbia, Maryland, April 19-21, 1982.

(11) Ross, D.T. and Schoman, K.T. "Structured Analysis for Requirements Definition", IEEE Trans. on Software Engineering, Vol. SE-3, NO. 1, January 1977, pp. 6-15.

(12) Keen, P., and Gambino, T.J., "The Mythical Man-Month Revisited", Proc. APL 1980, pp. 630-648.

(13) Carey, T.T. and Mason, R.E.A., "Information Systems Prototyping: Techniques, Tools, and Methodologies," INFOR. Canadian Journal of Operational Research and Information Processing, to appear.

(14) Concise Oxford English Dictionary.

A CONCEPTUAL FOUNDATION FOR VIEW INTEGRATION

C. BATINI, M. LENZERINI
Istituto di Automatica - Università di Roma
Via Buonarroti 12 - 00185 Roma - Italy

Abstract

View integration is a critical activity in data base design. Several methodologies for view integration have been proposed in the last years that afford the problem with different strategies and in the context of different data models. In this paper a general framework for comparing existing approaches and giving a conceptual foundation to the area is proposed. Within a model independent approach we investigate the activities involved in the integration process, in terms of semantic checks that are to be performed, types of restructuring that are usually needed and types of procedurality that can be chosen.

CONTENTS

1. INTRODUCTION

In recent years a lot of effort has been done to provide effective research guidelines for conceptual data base design methodologies (see [13],[14],[16]).Conceptual design of a data base is usually seen as divided into two steps:

- *view modelling*, during which user requirements are formally expressed by means of several user conceptual schemata
- *schema integration* (or *view integration*), that merges such schemata into a unique global schema of the application.

The design of the n user schemata may be in general developed independently, by different analysts and at different times. As a consequence, several complex tasks are to be managed during integration: finding the common parts between the different schemata, finding the different representations chosen by the analysts, in case discover inappropriate or unreliable choices; finally, discover interschema properties, i.e. properties involving data belonging to different schemata that were hidden to the analysts in former design steps.

The topic of schema integration has been recently addressed in several papers (see [3],[7],[8],[10],[11],[15],[17], [20]) that give different answers to issues pointed out in [13],[14],[16]). Practically, all those papers concern only with *data* integration, and do not address the topic of integration of dynamic aspects of the application. While such papers are important contributions to the problem, we believe that it is now the moment of developing, together with new ideas, a general framework for comparing existing approaches and the topics pointed out. In developing such

4.

investigation we have three goals:

1. find criteria of classification and comparison of existing methodologies for view integration.
2. give a conceptual foundation to the area of view integration, describing concepts and activities typical of the area without referring to any particular data model and methodology.
3. provide general guidelines, in a moment in which the research in the field is at a mature stage, for a schema integration methodology "parametric" with respect to the conceptual model.

The paper is organized as follows.
In Section 2 we develop a general framework to view integration, introducing several concepts that globally provide a model independent approach to this topic .
In Section 3 we analyze in detail the activities involved in the integration step in terms of semantic checks that are to be performed, types of transformations that are usually needed and types of procedurality that can be chosen.
In Section 4 we examine future research perspectives for this area.

2. A MODEL INDEPENDENT APPROACH TO VIEW INTEGRATION

In order to develop a general framework to view integration we need to introduce several concepts.

Data base design consists of a process of representation of a piece of the real world of interest (*Universe of Discourse*, UoD) on a computing machine.

Conceptual design is the phase of data base design in

which the UoD is formally described independently from the implementation environment (see [13]). Such a phase involves both static and dynamic aspects of the UoD, i.e. data, operations and events. In this paper we'll deal only with data design, whose goal is to obtain a formal description of data, called conceptual schema.

A *data model* may be seen as the formal language in which the conceptual schema is expressed; it consists of a set of structures in terms of which the objects of the UoD are described. Following the approach of [18], the allowed structures of a data model are specified in two complementary ways: classification structures and integrity constraints.

Classification structures are the structures by which the objects of the UoD are classified on the basis of common properties, giving raise to the concepts (or classes) of the conceptual schema. A class represents a set of objects of the UoD, called the *instances* of that class. For example, if *entity type* is a classification structure of the selected data model, the entity type EMPLOYEE, representing the class of persons employed in a certain enterprise of the UoD, may be a concept of the corresponding conceptual schema; each employee is an instance of the entity type EMPLOYEE.

Integrity Constraints are the structures that allow to specify rules on the concepts of the conceptual schema, reflecting semantic constraints on the corresponding objects of the UoD. For instance, if in the UoD the employees cannot earn more than their manager, an integrity constraint may be defined on the concepts EMPLOYEE, SALARY and MANAGER in order to represent the fact that the salary of an employee must be lower than the salary of its manager.

Two conceptual schemata are *equivalent* if they can

6.

represent the same universes of discourse.

The above definition of *equivalence* is obviously not constructive: another definition will be given in Section 3.

Union of *two* (n) *Universes of discourse* is the Universe of discourse whose things and happenings are the union of things and happenings of the two (n) universes of discourse.

Coming to a definition of *integration*, it is clear that we need a definition that does not distinguish between equivalent schemata.

Given two Classes of Equivalence of Conceptual Schemata C_1, C_2 (the definition is obviously extensible to n classes) their *Integration* $I(C_1, C_2)$ is the class of equivalence (of schemata) that represents the Universe of Discourse union of the two Universes of discourse represented by the given classes. In the following we will also speak of *integration of schemata* as an obvious extension of integration of classes of equivalence.

In terms of the above definitions we may say that the main role of a view integration methodology is to describe a way to obtain I from C_1, C_2 without repeating the entire conceptualization process for the Universe of Discourse $U \circ \mathcal{D} = U \circ \mathcal{D}_1 \cup U \circ \mathcal{D}_2$, the Universes of Discourse from which C_1, C_2 are derived. The reason for assuming $U \circ \mathcal{D}_1, U \circ \mathcal{D}_2$ as input for the design process comes indeed from organization constraints and from an hypothesis of "linguistic homogeneity" that can be made only within users, documents, etc. that describe each of the Universes of Discourse.

The tasks of such a methodology can be very complex. The reason for this comes from the following observations.

Assume for the moment that objects and properties of objects of the part of the $U \circ \mathcal{D}$ common to $U \circ \mathcal{D}_1$ and $U \circ \mathcal{D}_2$ have

been modelled exactly in the same way (i.e. by means of the same names, classification structures, and integrity constraints allowed in the model) in C_1 and C_2. We call this *assumption on the design, strong cohesion between schemata.*

Strong Cohesion may be lost in the design for several independent reasons (we refer to this situation as *Weak Cohesion Assumption*):

1. In the model several *equivalent representations* exist for the same Universe of Discourse (*Lack of Model Orthogonality*) (see an example in fig. 1, where dotted lines describe identifiers and symbols 1,1 and 1,n describe minimum and maximum cardinalities of instances of entities involved in relationships [7].

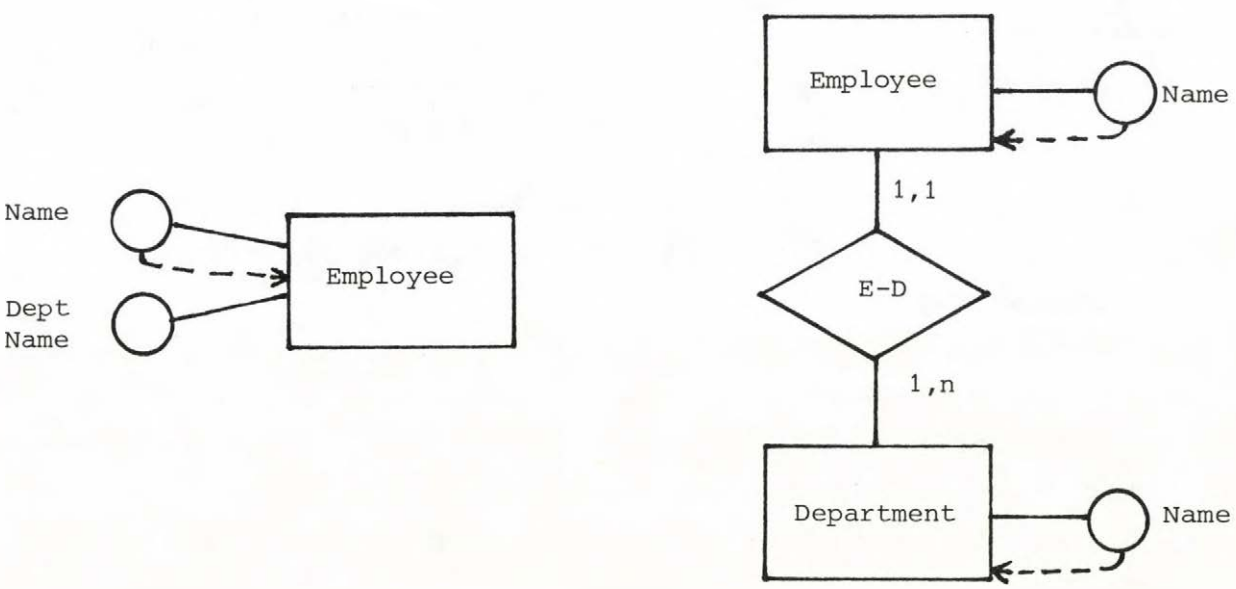


Fig. 1: *Example of Lack of Model Orthogonality.*

2. In the design process, different perceptions may have been adopted by different designers in modelling the

8.

same objects (*Pluralism of perceptions*). See for instance fig. 2, where the relationship between Employee

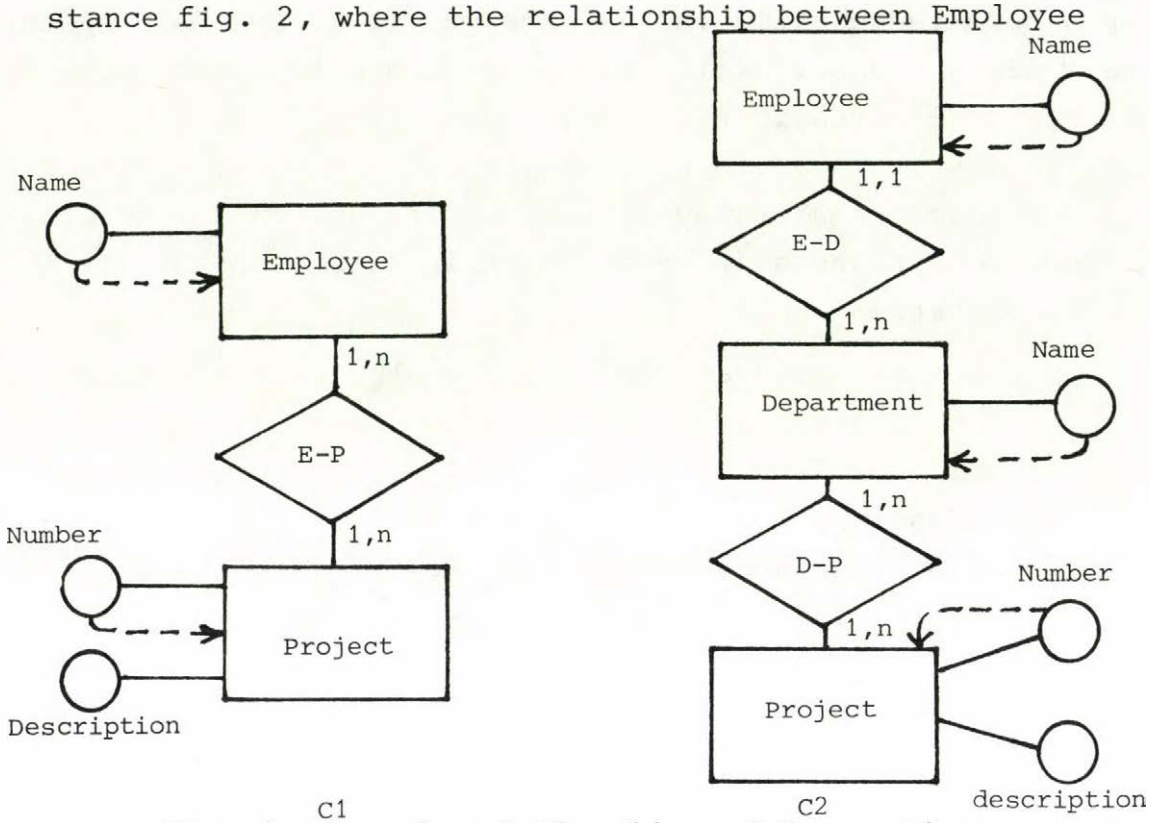


Fig. 2: *Example of Pluralism of Perceptions.*

and Project is explicitely perceived by the designer in C_1 , while in C_2 is implicitely perceived through entity Department.

3. In the design process, different abstraction levels may have been chosen to represent objects that belong to the same classes (*Heterogeneity of Abstraction Levels*). See for example fig. 3, where entity Person in C_2 is at a higher abstraction level with respect to entity Employee in C_1.

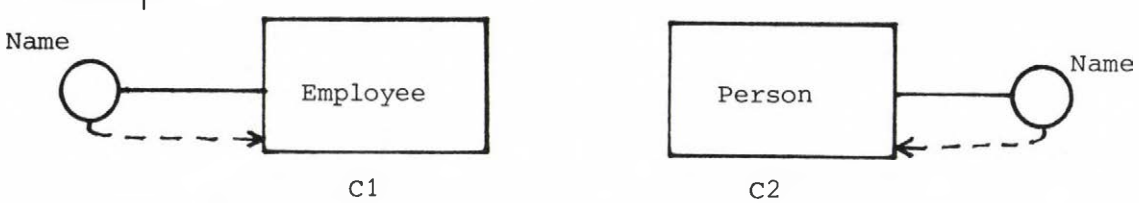


Fig. 3: *Example of Heterogeneity of Abstraction Levels*

9.

4. Several erroneous choices may have been made in the schemata for names, classification structures, integrity constraints, so that the conceptual design applied to UoD_1 , UoD_2 did not produce as a result the "true" schemata C_1 and C_2 but two schemata C_2',C_2' not equivalent to them (*Lack of Design Reliability*). See for example fig. 4, where it has been erroneously assumed in C_1 that an employee must be assigned to a unique project.

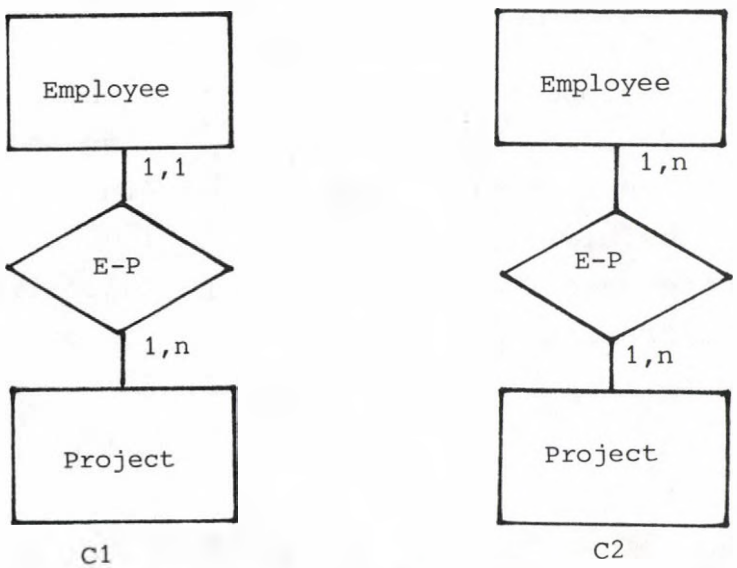


Fig. 4: *Example of Lack of Design Reliability.*

A "good" methodology for view integration should provide strategies to manage all of the above situations. In the next section we go deeper into the analysis, putting in evidence the activities and the conceptual cathegories involved in the integration process independently from the particular data model chosen in the methodology.

10.

3. ACTIVITIES AND CONCEPTS INVOLVED IN VIEW INTEGRATION

Topics pointed out in section 2 are afforded by existing methodologies for view integration with different strategies. Abstracting from specific proposals, we may single out the following concepts as peculiar of view integration.

a. Several *semantic checks* are to be performed by the designer in order to gain complete visibility on the meaning of the concepts in the schemata.
   In order to support such investigation, several types of *indications* can be considered, i.e. suggestions based on heuristics that guide the designer in its activities.
b. Several possible *transformations* are logically related to semantic checks and corresponding indications.
c. Several different *types of procedurality* can be proposed to perform the above checks.

In the following, we analyze in detail each of the above concepts.

3.1. *Semantic checks*

In comparing the schemata to be integrated, two different activities can be distinguished:

a. *Conflicts analysis*, whose goal is to find and conform the parts of the schemata representing the same piece of the *UoD*.
b. *Interschema properties analysis*, that looks for hidden properties between concepts belonging to different schemata.

In the following, we describe the characteristics of

these activities, assuming that the integration of two user schemata (S1,S2) is to be carried on. Furthermore we analyze the concepts of indication and scenario, that are useful to support the designer in semantic checks.

3.1.1. *Conflicts analysis*

The goal of this activity is to find all the concepts that are common to S1 and S2, and conform their representation.

The structure of this activity is the following:

INPUTS : S1,S2

OUTPUTS : $SS1 \subseteq S1$, $SS2 \subseteq S2$

such that: $SS1 = rep(x) \quad x \subseteq U_o\mathcal{D}_1$

$SS1 = rep(x) \quad x \subseteq U_o\mathcal{D}_2$

where:

- x is the maximum subset of $U_o\mathcal{D}_1 \cap U_o\mathcal{D}_2$ represented both in S1 and S2.
- $S = rep(U)$ means that conceptual schema S is a representation of the Universe of Discourse U.

During this activity all types of *conflicts* among the representations of the same objects in the schemata to be integrated are to be discovered. Conflicts may be classified as:

- naming conflicts
- structural conflicts

Naming conflicts

Let's call S1 ∩ S2 the schema obtained considering

those concepts that have the same *name* in S1 and S2.

Since under weak cohesion assumption SS1 and SS2 are in general different from S1 ∩ S2, this activity can be a very complex one.

The reasons for such difference come from *naming incoherences* between S1 and S2 and lack of design reliability, e.g.:

- *interschema homonymies* between S1 and S2 that imply the presence in S1 ∩ S2 of concepts representing objects that do not belong to $UoD_1 \cap UoD_2$.
- *interschema synonymies* between S1 and S2, that imply that objects in $UoD_1 \cap UoD_2$ are not represented in S1 ∩ S2.
- *intraschema homonymies or synonymies* (due to lack of design reliability) with analogous consequences.

Most of the methodologies mention naming incoherences, while only some of them ([7],[11],[15]) give specific guidelines to detect and solve them. Intraschema incoherences are mentioned only in [7]. In [3] the Universal Relation Assumption [6] is implicitely assumed: this assumption, in data base design, implies the absence of naming incoherences.

Structural Conflicts

When finding the common part, an activity of *comparison of information content* of the schemata has to be performed. Such comparison can involve conceptual structures at different level of granurality, i.e. atomic concepts, simple fragments, or even the entire schemata. E.g. in [8] it is suggested to compare pairs of concepts, while in [15] a comparison activity is performed on *subviews*, that correspond to simple fragments of the schemata.

Under weak cohesion assumption, several possible relationships may hold among conceptual structures repre-

senting the same piece of UoD: we call them *Equality*, *Equivalence*, *Containment*, *Compatibility*.

Equality

The structures are equal if the piece of the UoD they represent has been modeled by means of the same names, classification structures and integrity constraints allowed by the model.

Equivalence and Containment

The structures are equivalent if, even though they are not equal, they have the same information content. The equivalence is related to the concept of lack of model orthogonality.

Several definitions of equivalence have been proposed in the literature in different contexts (see for instance [5],[6],[12]).

We assume here a definition based on an approach appeared in [1].

Informally speaking, we can say that a schema S1 is less informative (<) than a second schema S2 if for every database i1 that is an instance of S1 a database i2, instance of S2, exists that has the same set of answers to queries. If S1 < S2 and S2 < S1, we say that they are equivalent.

The above definition provides a framework also for a concept of *information containment*.

The equivalence concept is explicitely used in [7] and in [15]. In [15] two different equivalences are taken into account, i.e. representation equivalence and restructure equivalence, defined on simple structures; the information on equivalent structures is considered as an input to the integration process. Equivalence is implicitely used in [3]

where, under the Universal Relation Assumption,the integration corresponds to merging sets of functional dependencies with the same closure.

With regard to the information containment concept, we notice that it is closely related to the activity of redundancy analysis, present in several methodologies ([3], [7],[11],[15],[20]).

In general, if a containment relation occurs between two structures belonging to different schemata, such containment gives raise to redundancy when the structures are put together in the integrated schema.

It is well accepted that redundancy analysis is a task of view integration: it is questionable [8] if all types of redundancies should be also eliminated during view integration.

Consider for instance the case of two paths in an Entity Relationship Model (see fig. 5) that are merged in

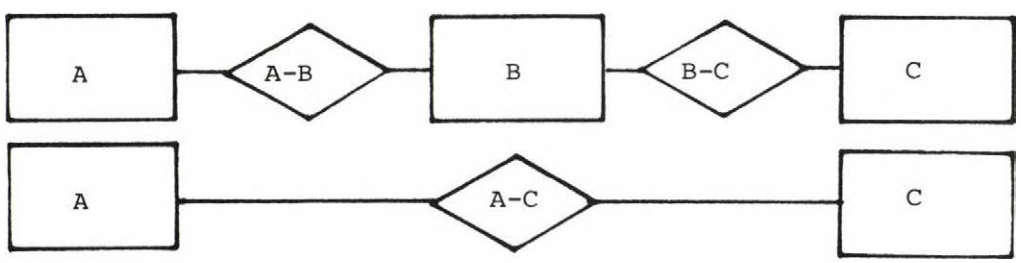


Fig. 5.

the integrated schema.

In case each of the relationships A-B,B-C,A-C can be derived by the remaining ones, no argument exists at conceptual design level to choose which of them to mantain in the integrated schema.

It is a task of physical or transaction design to choose either to represent redundant paths in the final

database schema and mantain them because of response time needs or to get rid of redundancy by selecting the more convenient path from the viewpoint of overall system's performance.

Compatibility

Quoting from [13] "a framework for rules needs to be developed to test views for merge compatibility".

What is compatibility? Intuitively, two conceptual structures are *compatible* if classification structures and integrity constraints referring to the same objects of the *UoD* are not contradictory.

In the view integration field the concept of compatibility and contradiction is strongly related with the so called *closed world* vs. *open world assumption.*

The closed world assumption states that a sentence on the objects represented in a conceptual schema is considered true if it is explicitely stated in the schema or is deducible from explicit sentences in the schema. In any other case the sentence is considered *false.*

In the open world assumption all sentences not stated in the schema or not deducible are considered unknown.

The closed world assumption is implicitely assumed in [10],where in the framework of the structural model two *entity relations* that represent the same object classes but have different attributes in the schemata are not considered mergeable; in order to superimpose them, two new subrelations are created in the integrated schema.

The open world assumption is implicitely assumed in [7],[11],[15] where pairs of compatible concepts are merged and the new concept in the integrated schema inherits all their properties.

16.

Examples of contradictions in the open world assumption are in [7] different min or max cardinalities for entities in the same relationships and in [15] different types of dynamic behaviour of concepts in the two schemata. Under open world assumption, contradictions are usually managed suggesting further investigation with the user.

3.1.2. *Interschema properties analysis*

Interschema properties are all the modelling features defined between different concepts in different schemata that were, as a consequence, *hidden* to the analyst in the design of a single schema.

Some of the interschema properties correspond to the discovering of a redundancy (see case 1 of fig. 6, where the redundancy is in the fact that all the instances of B are also instances of A);as we said in the previous section, in this case they reflect the presence of an information containment relation between fragments of the two schemata. Other interschema properties simply reflect new properties

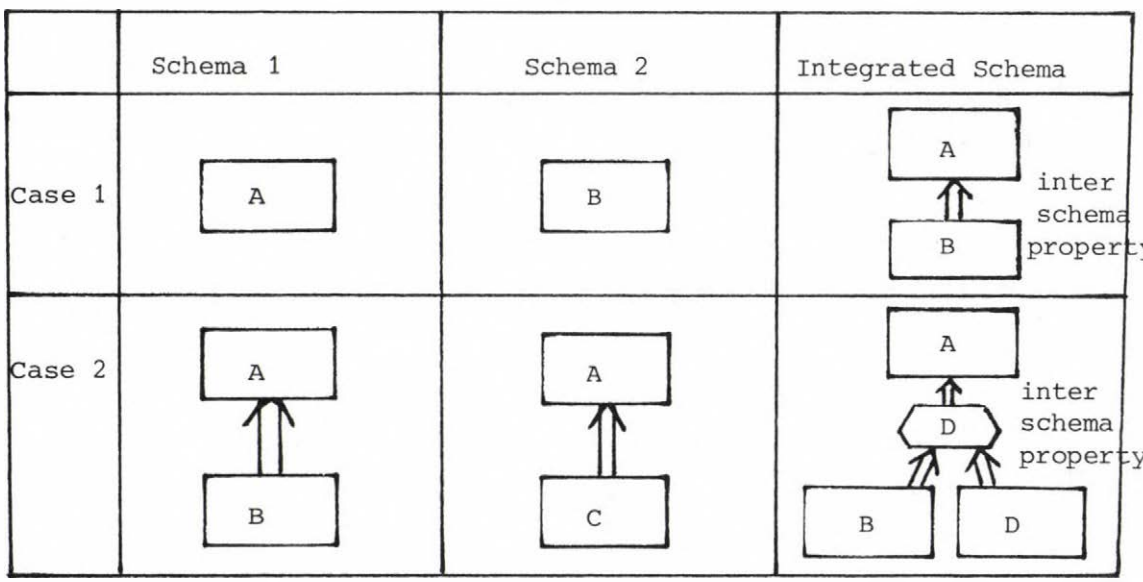


Fig. 6.

that are needed to gain completness in the representation of the global *UoD* (see case 2 of fig. 6).

In [17] it is shown the strong influence of the assumption of Heterogeneity of Abstraction Levels in the discovering of interschema properties.

In [10] interschema properties are used for gaining completness as well as constructing external views that have the property of being a proper subschema of the global schema.

Interschema properties are managed also in [7],[15]. At the end of this section we compare in Table 1 the design assumptions introduced in section 2 with the above described design activities, attempting an evaluation of their influence on such activities.

	Lack of Model Orthogonality	Pluralisms of Perspectives	Heterogeneity of Abstraction Levels	Lack of Design Reliability
Conlfict Analysis	si	si	wi	si
Interschema Properties Analysis	wi	si	si	si

si = strong influence

wi = weak influence

Table 1.

3.1.3. *Indications and scenarios*

Generally, a methodology should provide guidelines in

order to perform effectively and efficiently the investigations outlined in the above sections: such guidelines can be provided in the form of *indications*, i.e. situations that reveal potential conflicts, guide the designer and control the combinatorial explosion of possible investigations. Several *factors* can influence the way in which indications are managed in a methodology:

1. *the conceptual model;* e.g. a rich linguistic capability to express integrity constraints can be used in evaluating the similarity of concepts that are potential synonyms.
2. *peripheral information,* collected by the methodology, i.e. information on the neighbours of the UoD that is not destined to be represented in the conceptual schema and is collected and used to make more reliable the analysis on data represented in the conceptual model. In [19], for instance, keywords are collected for each concept, that represent a meaningful subset of its "neighbour concepts". Two concepts are considered potential synonyms if most of their keywords are equal.
3. *linguistic heterogeneity* between users, conventions, standard documents of the different subsystems of the organization.

At present, indications are dealt with in the methodologies for several goals.

In [11] *similarity* indications are suggested for discovering interschema naming incoherences. In [7] *concept likeness/unlikeness* are suggested for interschema incoherences and interschema properties, and *multiname anomalies* for intraschema incoherences; *new cycles occurrences* (in

the integrated schema) are suggested for redundancy analysis. In [15] *view similarity* is used for compatibility analysis.

Indications are potential motivations for some more investigation with the user: the investigation can lead the designer to discover either a conflict or an interschema property. As a consequence, several alternative modifications to the schemata are logically related to an indication; a methodology should suggest, for every specific indication, several corresponding *scenarios*, i.e. the type of conflict or interschema property the indication, in the given context, potentially reveals and the related modification to the schemata. Methodologies differ in the way in which they manage scenarios. Most of them suggest usually only one scenario: e.g. in [14], when an incompatibility is discovered between views, as a general policy the integrated view will include the more constrained one. In [7] usually several scenarios are suggested.

3.2. *Transformations*

When the schemata object of the integration process are analyzed, several possible transformations are needed that change some part of the schema in a new one.

Transformations can be classified in several ways. We propose here a classification (see fig. **7**) based on the definition of equivalence we have given in section 3.1.

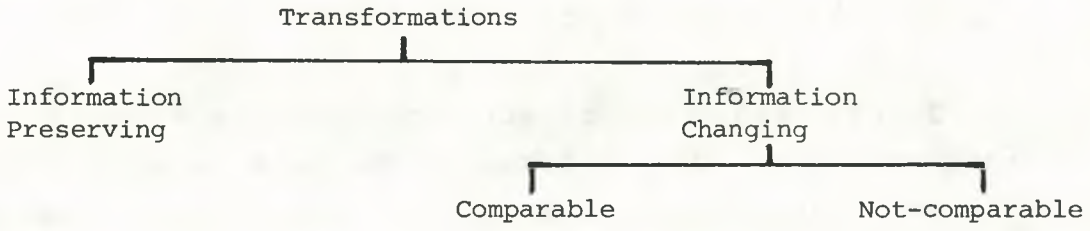


Fig. 7

20.

Information preserving transformations occur when for some design goal the designer aims at changing the syntactic representation of the schema, without changing its information content.

In [7], several equivalence transformations are suggested in order to unify types of concepts with the same name in the two schemata and simplify further design decisions.

Information changing transformations can be classified in:

- *Comparable transformations* when the information content of the two schemata can be compared, i.e. we can say that the previous schema is more (less) informative than the restructured one.

In [15] several "enhancement operations" are suggested that enrich the information content of schemata. In [7] when incompatible representations appear in the two schemata for concepts with the same name, one of the two concepts is modified. Similar transformations appear in [10],[15],[20] in a phase of the analysis that concerns concepts in the integrated schema.

- *Not comparable transformations* are usually needed when owing to previous insufficient or unreliable design, conflicts arise (e.g. homonyms or synonyms) that must be solved with a renaming or change of the structure.

Existing methodologies have different approaches in suggesting when affording transformations.

In [10],[15],[20] firstly schemata are merged and then transformations are performed on the integrated schema. In [7] some transformations an performed on input schemata and

others on the integrated schema.

Notice that at the end of the integration process it is usually convenient to perform further transformations on the integrated schema for goals different from those examined till now, i.e.:

1. express as far as possible by means of the model itself all the integrity constraints otherwise expressed by means of natural language. We call *autoexplicativity* this quality of the design.
2. gain further *clarity* and *simplicity* in the representation of the *UoD*.

Similar goals are also typical of the view modelling step of conceptual design; the analysis can be reproposed now for two different reasons:

1. this is the final step of conceptual design, and so it is crucial at this point to gain high quality of the design.
2. only at this phase of the design it is possible to get a centralized view of the global *UoD* of interest for the application.

3.3. *Types of procedurality*

Several *types of procedurality* can be used in a view integration methodology, corresponding to the different choices that the designer has at is disposal in creating a partial ordering between the different design steps that are to be performed.

They may concern:

1. Linguistic transformations
2. Design strategies
3. Order of integration between schemata

4. Order of integration between concepts to be merged
5. Order of integration between modelling structures.

In the following we analyze the above procedualities.

3.3.1. *Linguistic transformations*

The fundamental goal of a Methodology for Data Base Design is to transform a user oriented linguistic representation (l.r. in the following) of requirements into a DBMS oriented one.

In order to simplify and make more reliable such transformation, the introduction of two intermediate phases in such transformation process is usually proposed (see [13],[14]), i.e., from the bottom the top:

1. a l.r. independent from the user and the DBMS, usually called *conceptual model*.
2. a l.r. independent from the user and from conceptual model, i.e. obtained from the initial requirements in such a way that no choice has to be made at this level, regarding the structures used to represent the information of interest. According to [14], we call this l.r., *Requirements Model*. We adopt the term *Requirements Schema* to indicate an instance of it.

See in fig. 8 a comprehensive representation of the above terminology.

Integration can be in principle performed at each of the above levels. For instance:

1. The enterprise schema mentioned in [14] can be considered as the result of a first integration step afforded before Conceptual Design (it is indeed usually considered

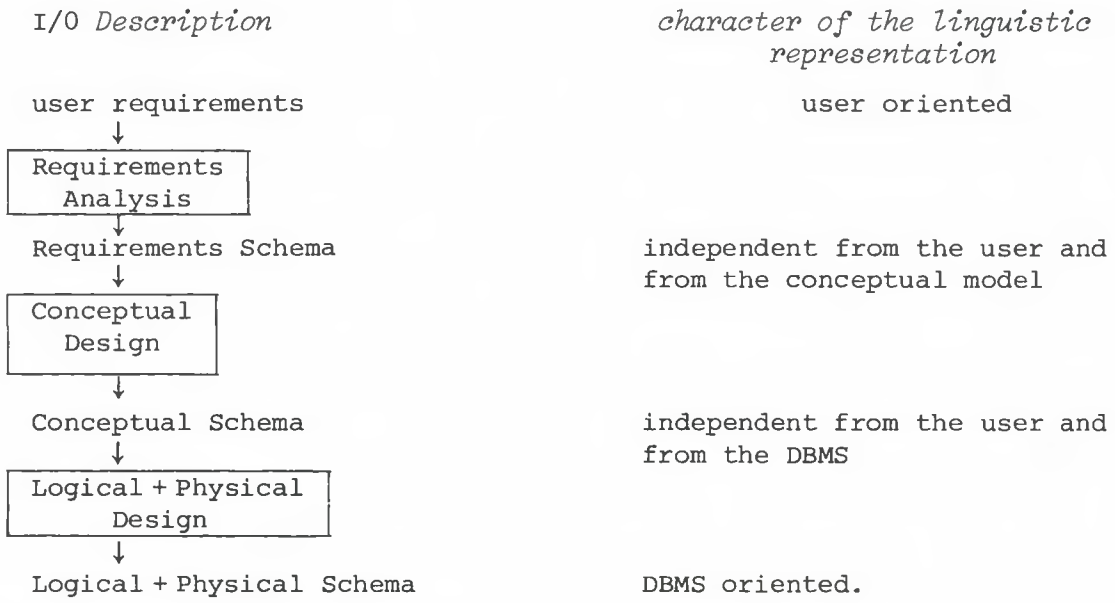


Fig. 8.

as an input to Conceptual Design).

2. integration can even be performed during logical design. Synthesis algorithms in [3],[4] are examples of such approach.

Most of the methodologies perform integration at the conceptual level. This approach can be considered as a tradeoff between two different requirements:

a. as in software design, in data base design too error cost increases dramatically during the life cycle of the application. This aspect should justify when possible an integration "in the head of the designer".
b. on the opposite side, due to the great complexity of the integration process, it seems better to perform such activity only when formal, unambiguous representations have been produced.

24.

3.3.2. *Design strategies*

As we pointed out in [2], in data base design we can use the terms "top-down" and "bottom up" to characterize the different strategies proposed in the literature for conceptual design.

For instance, the refinement of an entity into a more complex structure that inherits its links in a conceptual schema can be considered as a top-down activity, while the integration of two schemata (or else of a new entity to a schema) is a bottom-up activity.

In principle, the designer of a conceptual schema should be allowed to intermix top-down and bottom-up activities. As a consequence (while methodologies usually propose two distinct and clearly specified activities for view modelling and schema integration) in general the integration step should be allowed at any level of refinement, in order to carry on the design intermixing view modelling and schema integration.

Most of the existing methodologies for view integration do not afford this problem: they assume that the view modelling process has been concluded so that the schemata to be integrated are assumed as specified at the final level of refinement.

Some basic concepts regarding to this aspect can be found in [17], where the proposed data model is based on abstraction mechanisms and general guidelines to integrate user views possibly specified at different levels of abstraction are provided.

3.3.3. *Order of integration between schemata*

This aspect involves two related problems: giving a

general strategy for the entire integration process in order to produce a global schema from several conceptual schemata and providing criteria for the choice of the order of aggregation of such schemata.

With regard to the first point, the concept of *integration tree* can be introduced: let's call $C_1,\ldots,C_n$ the schemata to be integrated (user schemata in the following) and CS the global conceptual schema.

The procedurality of the integration process can be represented by means of a tree according to the following rules:

- the root represents the global schema CS
- the leafs represent user schemata $C_1,\ldots,C_n$
- the intermediate nodes represent partial integrated schemata
- for each node, its children represent schemata from which it has been derived by means of an integration step.

Stating the structure of the integration tree corresponds to provide the general strategy to accouplish the integration process. Most of the methodologies, for example, agree in adopting a binary tree because of the increasing complexity of the integration step with respect to the number of schemata to be integrated.

The proposals in [3],[20] can be considered exceptions to this rule: n-ary integration steps are allowed in their approaches in which however, the types of conflicts and situations taken into account in the analysis is quite limited.

With respect to the balancing of the integration tree, two alternative choices have been proposed: respectively a

completely balanced [17] and a completely unbalanced binary tree [7].

In [17] it is argued that the balancing of the integration tree minimizes the number of comparisons between concepts of the schemata that are performed at intermediate steps in the integration process. In the approach of [7] the integration of schemata with higher relevance is anticipated so to obtain a better convergence and stability in the construction of the partial integrated schema.

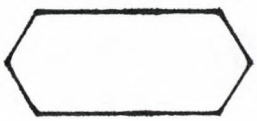
3.3.4. *Order of integration between concepts to be merged*

This aspect and the next one is meaningful when a procedurality has been chosen for the integration of schemata, and two or more schemata are to be integrated in a new one. At this stage, in order to discipline the explosion of possible activities, at least two different strategies can be chosen.

A first class of strategies proceed imposing an order to classification structures allowed in the model, and integrating in such order the corresponding "layers" of the schemata.

A possible criterion for the choice of the order should tend to anticipate as soon as possible the most critical choices, achieving fastly a first convergence of the design.

We show, for example (see fig. 9), the metaschema of an Entity Relationship Model [9] enriched with subset and generalization abstractions for entities (Sub and Gen relationships) and min and max cardinalities. We assume the metaschema selfexplanatory, except for symbol:



called underlying attribute, i.e. the attribute of the entity at the upper level in the generalization whose values correspond to names of entities at lower level.

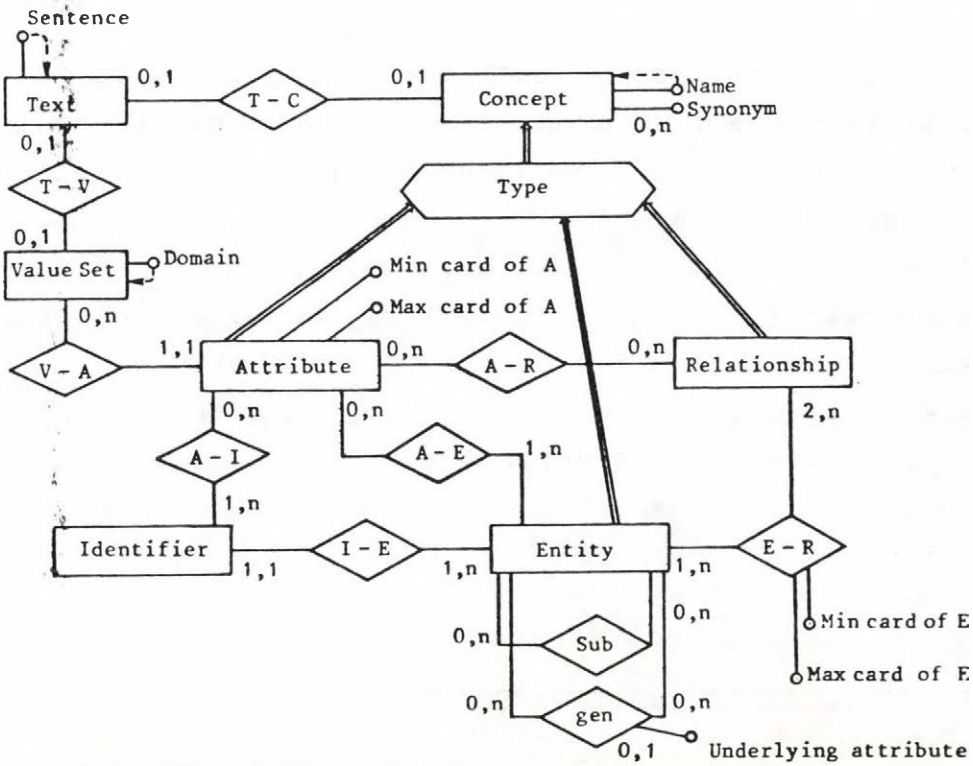


Fig. 9: Entity-Relationship Model Metaschema.

Since in such a model the entity concept is the most significant one, if this procedurality and this model are assumed it is useful to anticipate entity analysis. In several methodologies[10],[15] this criterion is widely applied.

3.3.5. *Order of integration between modelling cathegories*

A second order that could be chosen with the goal of finding layers of the schemata to be subsequently integrated, is based on modelling cathegories i.e.:

a. names
b. classification structures
c. integrity constraints.

The idea here is that the naming activity is the most primitive one when a Universe of Discourse is conceptualized in a schema. As a consequence, when two or more schemata are integrated, first of all names of concepts are unified independently from classification structures and integrity constraints chosen for modelling there. Subsequently, classification structures of the concepts with the same name are analyzed, attempting to unify them according to transformations that preserve equivalence. Finally, integrity constraints are analyzed in order to check their compatibility. Such an approach is chosen in [7].

4. CONCLUSIONS AND FURTHER RESEARCH

In this paper an attempt was made to develop a general framework and give a conceptual foundation to the area of view integration.

Research and practical experience are needed to compare existing methodologies and integrate the most effective approaches for single activities.

Furthermore, tools are to be developed that support the designer in suggesting indications and scenarios and perform transformations. Moreover, the integration of dynamic aspects has to be afforded; this aspect is practic-

ally ignored in existing methodologies.

ACKNOWLEDGMENTS

The authors wish to thank Giovanni Vocalelli for useful discussions and suggestions.

REFERENCES

[1] P. ATZENI, G. AUSIELLO, C. BATINI, M. MOSCARINI: Inclusion and equivalence between relational database schemata. Theoretical Computer Science 19, 1982.

[2] P. ATZENI, C. BATINI, M. LENZERINI, F. VILLANELLI: INCOD-DT: A System for Conceptual Design of Data and Transactions in the Entity - Relationship Model, Proc. 2nd Int. Conf. on the Entity Relationship Approach, Washington, 1981.

[3] S. AL-FEDAGHI, P. SCHEUERMANN: Mapping considerations in the design of schemas for the relational model. Techn. Rep. N. 79-05 Northwestern University, Illinois.

[4] P. BERNSTEIN: Synthetizing third normal form relations from functional dependencies. ACM Trans. on Database Systems, vol. 1, N. 4, 1976.

[5] M. BILLER: On the equivalence of Data Base Schemes: a semantic approach to data translation. Information Systems, Vol. 4, 1979.

30.

[6] C. BEERI, P. BERNSTEIN, N. GOODMAN: A Sophisticate's introduction to database normalization theory. Proc. Conf. on Very Large Data Bases, 1978.

[7] C. BATINI, M. LENZERINI: A methodology for data schema integration in the Entity-Relationship Model. Technical Report R82.08 Istituto di Automatica, 1982.

[8] C. BATINI, M. LENZERINI, M. MOSCARINI: Views Integration, in S. Ceri (ed.): Methodology and tools for data base design. North-Holland 1983.

[9] P. CHEN: The Entity Relationship Model: Toward a Unified View of Data. ACM Trans. on Data Base Systems, Vol. 1, N. 1, 1976.

[10] R. EL MASRI, G. WIEDERHOLD: Data model integration using the structural model. Proc. ACM Sigmod Int. Conf. Boston, 1979.

[11] B. KAHN: A Structural Logical Data Base Design Methodology PhD Thesis,University of Michigan, 1979.

[12] P. KANZIA, H. KLEIN: On the equivalence of databases in connection with normalization. Proc. Int. Workshop on Formal Bases for Data Bases, Toulouse 1979.

[13] V. LUM et al. 1978: New Orleans Data Base Design Workshop. Report Proc. 5th Int. Conf. on Very Large Data Bases, Rio de Janeiro, 1979.

[14] S. NAVATHE et al.: Information Modelling tools for Data Base Design. Data Base Directions,Fort Lauderdale, 1980.

[15] S. NAVATHE, S. GADGIL: A methodology for view integration in logical data base design. Proc. 8th Int. Conf. on Very Large Data Bases, Mexico City 1982.

[16] Proc. New York Symposium on Database Design, New York 1978.

[17] T. TEOREY, J. FRY: Design of Database Structures, Prentice Hall 1982.

[18] D. TZICHRITZIS; P. LOCHOVSKY: Data Models. Prentice Hall 1982.

[19] M. TARDIEU, D. NANCI, D. PASCOT: Conception d'un systeme d'information - construction de la base de données, Edition d'organization Paris 1979.

[20] S. BING YAO, V. WADDLE, B. HOUSEL: View Modelling and Integration Using the functional data model. IEEE Trans. on Software Engineering, Vol. SE8, N. 6, 1982.

SAS - A SPECIFICATION SUPPORT SYSTEM

Michel LISSANDRE, Pierre LAGIER, Ahmed SKALLI

I G L — Institut de Génie Logiciel — Paris, France

◆◆◆◆◆◆◆

ABSTRACT

The software crisis has shown the utmost importance of building up any system development on "good" specifications.

SADT*, the well-known Softech's graphical method has been applied to a broad spectrum of complex system engineering problems. Perhaps because of its graphical aspect, the method is not currently fully automated by a tool which would be at once complete, accessible, portable, and which would support the variety of ways to practice SADT.

SAS, tool currently developed by IGL aims at becoming such a tool. SAS allows to create and edit diagrams, to build "kits", sets of diagrams which, by the bringing of a reader/author cycle procedure into operation, assure quality to implement this reader/author cycle, to maintain the project files, to perform some syntactic and semantic checks, along with quality and productivity measurements.

The developments of this tool, being of the incremental type, has led to a first release in February 1983 and will be pursued until 1985.

---

* SADT is a trademark of Softech, USA & IGL, France.

1. BACKGROUND

The cost of software development and maintenance, the impact of analysis and design errors, and the shortcomings of traditional analysis and design approaches have brought about a software crisis.

Until recently, software professionals responsible for analysing users' requirements and designing computer based systems have had to spend a significant amount of time creating at once the analysis and design approaches to be used, the environment in which to bring them into operation, as well as solving the technical problems at hand.

Sofware engineering is a disciplined and controlled approach that deals with many key problems associated with software development. The frame-work is established by a uniform system life cycle that incorporates the standard procedures and documentation that are necessary for analysis, design, and implementation activities. Underlying this standardization is the successful application of a variety of tools and techniques that take advantage of the principles of structuring.

During the last decade, research has concentrated on techniques for defining, analysing, and documenting the requirements for systems. Many of these techniques complement other structured techniques in the system development life cycle such as structured programming. The objective of structured analysis is to provide a methodical approach for documenting system requirements and analysing the integrity of the requirements. The essential purpose is to thoroughly evaluate the needs and requirements which the system must satisfy, commonly called problem definition, preceding the actual design and implementation phases of the system development life cycle.

Manual structured analysis techniques are very effective for small problem definition activities. The magnitude of the creation, maintenance, and review of large problem definitions has made it practical to develop computer-aided support tools.

However, the existing tools deserve one or more remarks among the following :

♦ They are dedicated to a particular type of software (process control, management,...) or to a particular type of specification (performance analysis, simulation, implementation, static description,...).

♦ They are only used by software professionals.

♦ There is only one way to operate them.

♦ The specifications they generate are difficult to understand and even read, which makes their control by non-specialists almost impossible.

Of course, these tools may be very useful in their own application domain, but it is a commonly made mistake to expect them to provide more than what they can do, or to fulfill a different purpose than the one they have been designed for.

The tool we shall present differs from those mentioned above in the following points :

♦ It must be easy-to-learn and easy-to-use, thus, non-specialists, and of course non-software professionals (customers, users, managers,...) will benefit of its use.

♦ It must cope with various ways of practicing the supported specification method. Specifying and designing complex systems is a creative process. Two analysts may then come to similar results by different means, even within the strict bounds of a rigorous, disciplined method. If the tool does not allow each analyst to follow his way of thinking, it will not be used, or will be misused by this analyst, with all the negative consequences.

♦ It must support the analysis and specification of all kinds of systems : software systems, embedded systems, but also systems where computers play only a marginal role, or even are completely missing : "P3 systems" (Paper, People, Procedures). It must allow to specify at various abstraction levels and from various viewpoints.

♦ It must – last but not least – provide a solution to the major problem of any specification : its legibility by people of various skills. It must allow and make it easier to communicate the specifications between all the partners involved in any big project. The specifications produced with this tool must be found as clear, readable, understandable, checkable and verifiable by the "upstream" intervening parties (customers, operators and various users, main contractor,...) as by the "downstream" ones (subcontractor, designers, implementers, maintenance teams,...).

2. S A D T

Softech has developed a methodology to deal with these problems, known as SADT (Structured Analysis and Design Technique) and, together with a few European companies (IGL being one of those for the French speaking European and African countries), offers licensing, training, and consulting assistance in its use.

SADT supplies its manual user with a good answer to the last three points referred to above.

Today, hundreds of projects and thousands of analysts have used SADT all over the world. Hence it is very likely that the reader of this paper already knows SADT. He/she should then skip this chapter where the major concepts will be briefly described, highlighting the features which will impose specific functions to a supporting tool.

Since its development, SADT has been presented in various papers [ROSS-76], [ROSS-77a], [ROSS-77b], [DICKOVER-77], [CONNOR-80] in which the reader can find additional information. This paper only gives a summary, highly based upon [ROSS-76] and [CONNOR-80].

SADT is a methodology developed by Douglas T. Ross in 1974 that is useful for system planning, requirements analysis, and system design. It was created to provide a rigorous, disciplined approach to achieve understanding of user's

needs prior to providing a design solution. SADT did not evolve from a design technique, but rather was developed by examining the problems associated with defining system requirements. It is generally not used for software module (program) detailed design because SADT does not contain the constructs necessary for program design (sequence, selection, and iteration).

SADT provides the user, the system analyst, and the system designer with a diagramming technique to structure the products of analysis and design, a set of methods to structure the procedures of performing analysis and design, and a set of management and human factors to structure the overall process of analysis and design.

There are seven fundamental concepts underlying SADT :

♦ Complex problems are best attacked by building a model which expresses an in-depth understanding of what the problem is and which is sufficiently precise to serve as the basis for the problem solution.

♦ Analysis of any problem should be top-down, modular, hierarchic and structured.

♦ The model should be represented by a diagramming technique which shows component parts, their interfaces, and how they compose a hierarchic structure.

♦ The model-building technique must represent both things (objects, documents or data) and happenings (activities performed by men, machines, computers, software). The model must show both aspects properly related.

♦ The analyst should differentiate as much as practicable between an initial functional model of functions to be performed and a subsequent design of model of how those functions will be performed.

♦ Analysis methods must support disciplined, coordinated teamwork.

♦ All analysis and design decisions and comments thereon must be in written form and available for open review by all team members.

A natural language is not precise enough to express requirements and system designs and to ensure cost-effective system development. Natural languages tend to be verbose, redundant, and subject to interpretation. Therefore, in order to take advantage of the principles of structuring, it is imperative that we should employ a graphic technique that focuses on displaying activities and data, allows the gradual introduction of detail, and is suitable for showing information in a top-down manner.

Using a graphic technique to explain requirements or a system design involves developing a model. A model is a representation of reality —an "expression of one thing we hope to understand in terms of another we think we do understand" [WEINBERG-75].

A SADT model is an organized sequence of diagrams. A high-level overview diagram represents the whole subject. Each lower-level diagram shows a limited amount of detail about a well-constrained topic. Further, each lower-level diagram connects exactly into the model to represent the whole system, thus preserving the logical relationship of each component to the total system (See figure 1).

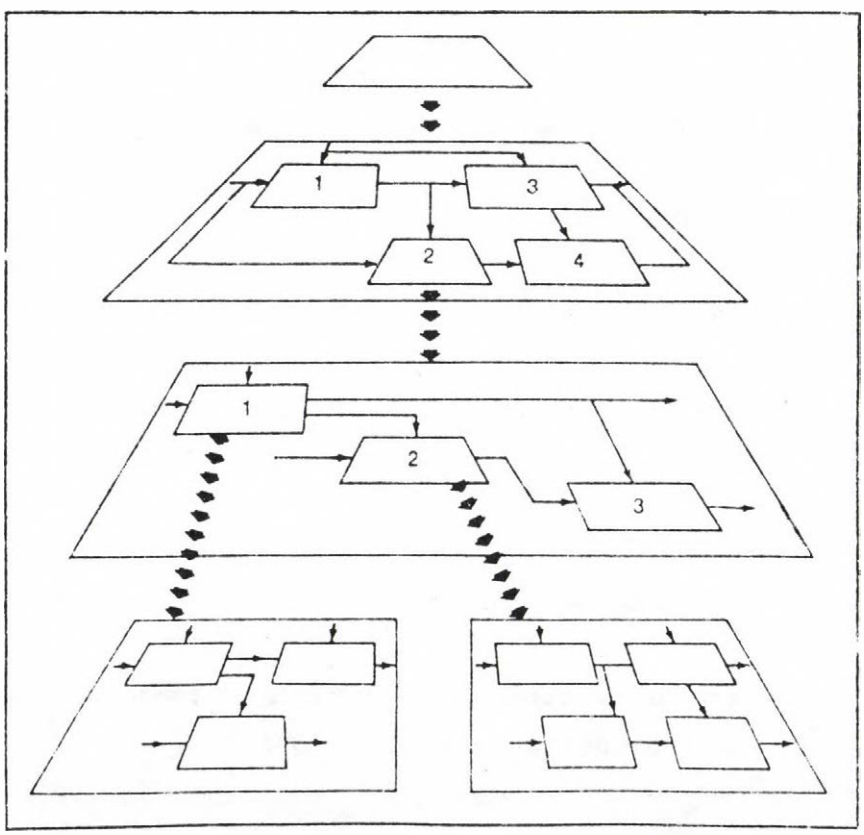


Figure 1 —Hierarchical Structure of SADT Diagrams

SADT analyses two major aspects of each system — its data and its activities. This is done by developing two complementary models, an activity decomposition and a data decomposition. The activity decomposition details the happenings as activity boxes, while showing the things that interrelate them as data arrows. The data decomposition details the things of the system as data boxes, with the happenings that interrelate them shown as activity arrows.

Each SADT model consists of diagrams made up of three to six boxes, and arrows. On an activity diagram within an activity model, the boxes represent activities, and the arrows represent data. It is just the opposite on data diagrams.

On an activity diagram, a box is named with a verb. The left-hand side of the box is used to show input data, labelled with a noun, to be transformed by the activity : the incoming data flow. The right-hand side of the box shows output data, which is data transformed by the activity that is to be used elsewhere, that is, outgoing data flow.

Unlike other diagramming techniques, SADT also describes control and supporting mechanisms. The top of the box is used to show control data, which are data that constrain the operation of an activity. This information has two major purposes. First, the distinction between input and control allows the system analyst or designer to explicitly show data that are not transformed into output, and therefore, are used to modify the behavior of an activity. Second, the introduction of control data allows the analyst or designer to evaluate the cohesiveness and functional representation of all of the boxes on a diagram. If all relationships were truly input/output, procedural coupling would be the only degree of strength that could be evaluated [MYERS-75]. Constraint relationships must be shown in order to distinguish between the degrees of binding and to allow a qualitative evaluation of the decomposition to be performed.

On an activity diagram, the bottom of the box is used to show a supporting mechanism of the activity. That is, if the analyst or system designer wishes to describe organizations that perform a given activity, the mechanism arrow is used to identify the department, section, or even the individual that is responsible for the activity. Another extremely important use of the mechanism side of the box is for cross-referencing models.

An example of a SADT diagram is shown in Figure 2.

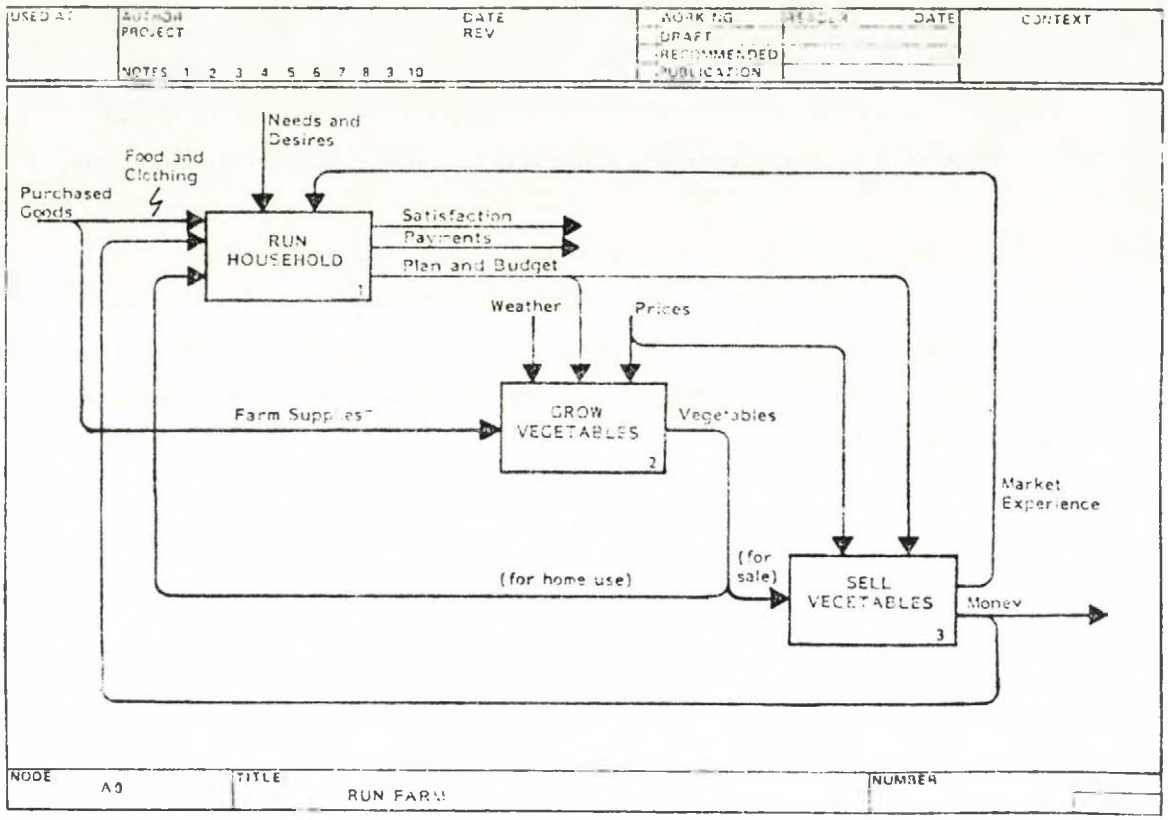


Figure 2 — SADT Activity Diagram

The final step in the modeling process is to tie together the activity and the data portion. Each decomposition is checked to make sure that its use of dual elements is coherent. This process reduces errors and oversights and assures consistency in further work.

No one person can completely understand every aspect of a complex system within the time limits usually imposed. Even if this were possible, it would place an undesirable demand upon one person. Analysis requires disciplined, coordinated teamwork. Consequently the insights and views of project personnel must be communicated effectively at every step and level of analysis to insure that the SADT models reflect the best thinking of the team. Adequacy and

quality must be assured by regular, critical review, so that changes and corrections can be made on an incremental evolutionary basis.

Because SADT starts with single black box and proceeds to increasingly detailed diagrams of elements of the problem, documentation becomes available on a continuous basis. At each step decisions can be seen in context and challenged while alternative approaches are available. The documentation provides the basis for decisions and vastly improves the visibility of the project to the team and to management.

Cooperative teamwork demands a clear definition of the types or interactions which should occur between the staff involved. SADT anticipates this need by establishing titles and functions of appropriate roles.

Throughout a project the draft versions of the diagrams produced are distributed to other project members for review and comment. SADT requires that each person making comments about a diagram will make them in writing and submit them to the author of the diagram. Such an approval cycle continues upward in the organizational structure until the diagrams and eventually the entire model are officially accepted.

A SADT librarian provides filing, distribution, record-keeping support, and precise control over the status of the evolving model. Since everything is on record, future enhancement and system maintenance can refer to previously-taken decisions.

3. **S A S** (from the French " Systeme d'Aide à la Spécification")

SAS takes advantage of the solutions brought up by SADT to the problem mentioned in chapter 1, mainly the communication problem, but provides its user with a set of additional facilities :

- training aids ;
- computer assisted drawing ;

- automatic controls and verification of :
  . conformance to SADT basic rules,
  . coherence of boxes, arrows and labels within a diagram,
  . coherence of diagrams within a model,
  . coherence of models within a project ;
- program design aids ;
- complexity measurements ;
- productivity measurements ;

SADT, as supported by SAS is the standard SADT as developed by Softech, and as used since. Its wide usage makes it now possible to consider SADT as a "standard". Because of this wide usage, SAS has been designed to be highly portable so that organizations using equipments ranging from top scale micros (on local networks or not) to big main frames, may benefit from its use.

The functions SAS performs are — charity begins at home —expressed by the SADT diagrams shown in Figure 3 to 7.

However, SAS' major points are being discussed in the following paragraphs.

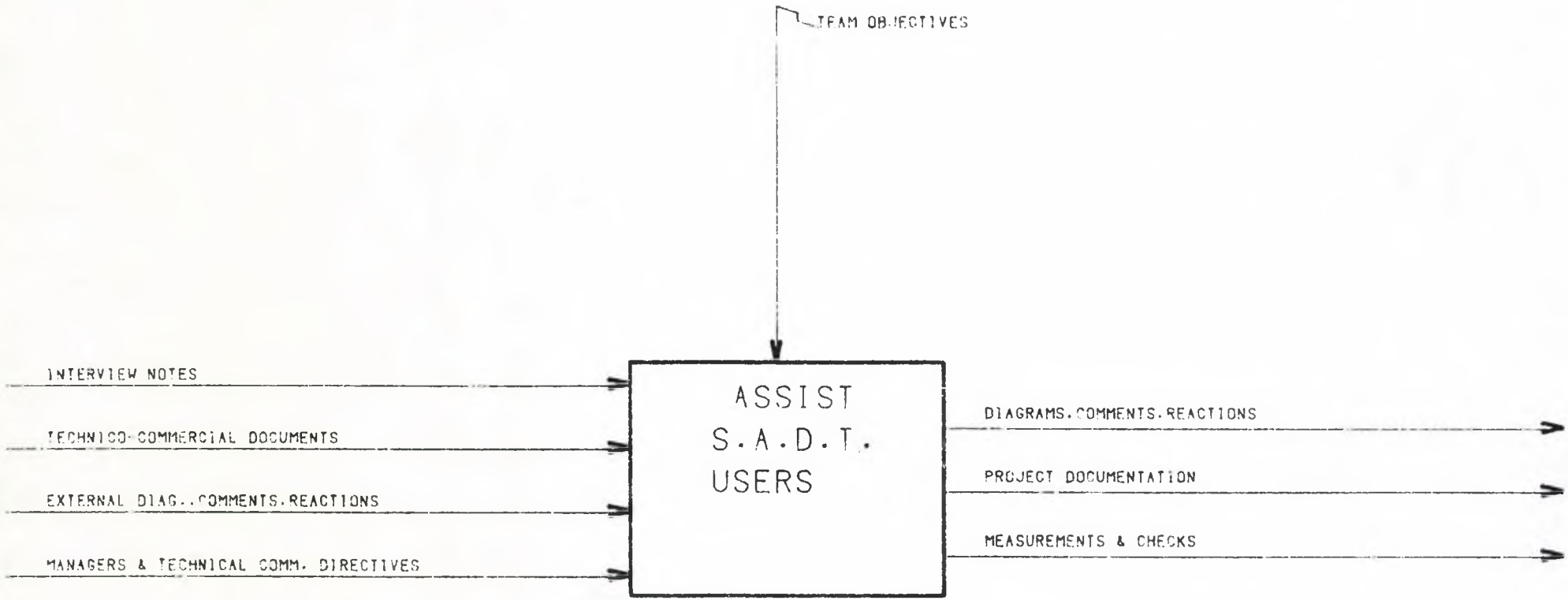
3.1. Creating and editing diagrams

This section deals with the functions represented by the following boxes of the model :

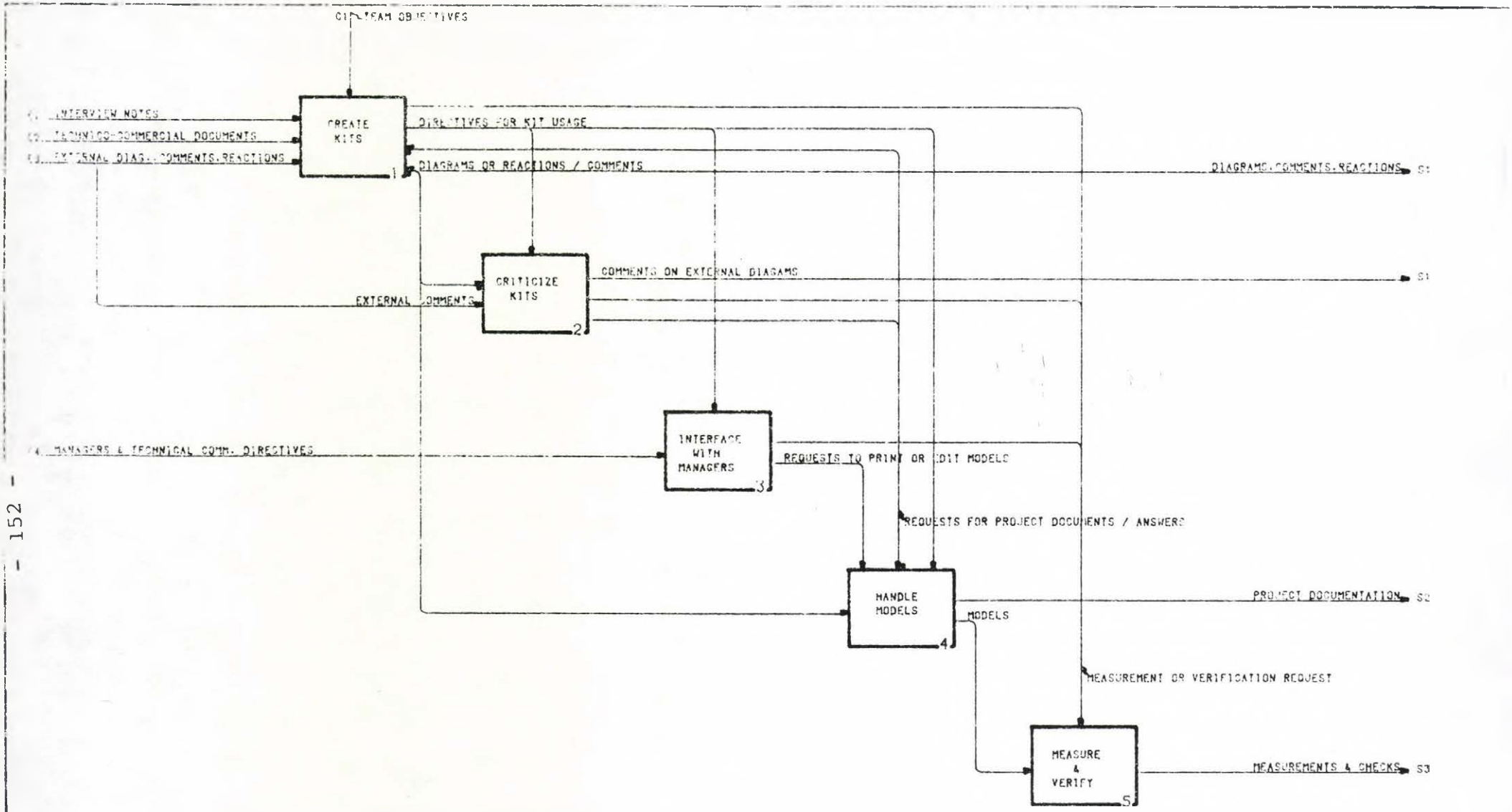
- Create kits (Diagram A0, box 2)
  (more specifically :
  . Create a new diagram (Diagram A1, box 2)
  . Obtain inputs from external authors (Diagram A1, box 3)
  . Update diagrams (Diagram A1, box 4))

- Criticize kits (diagram A0, box 2)

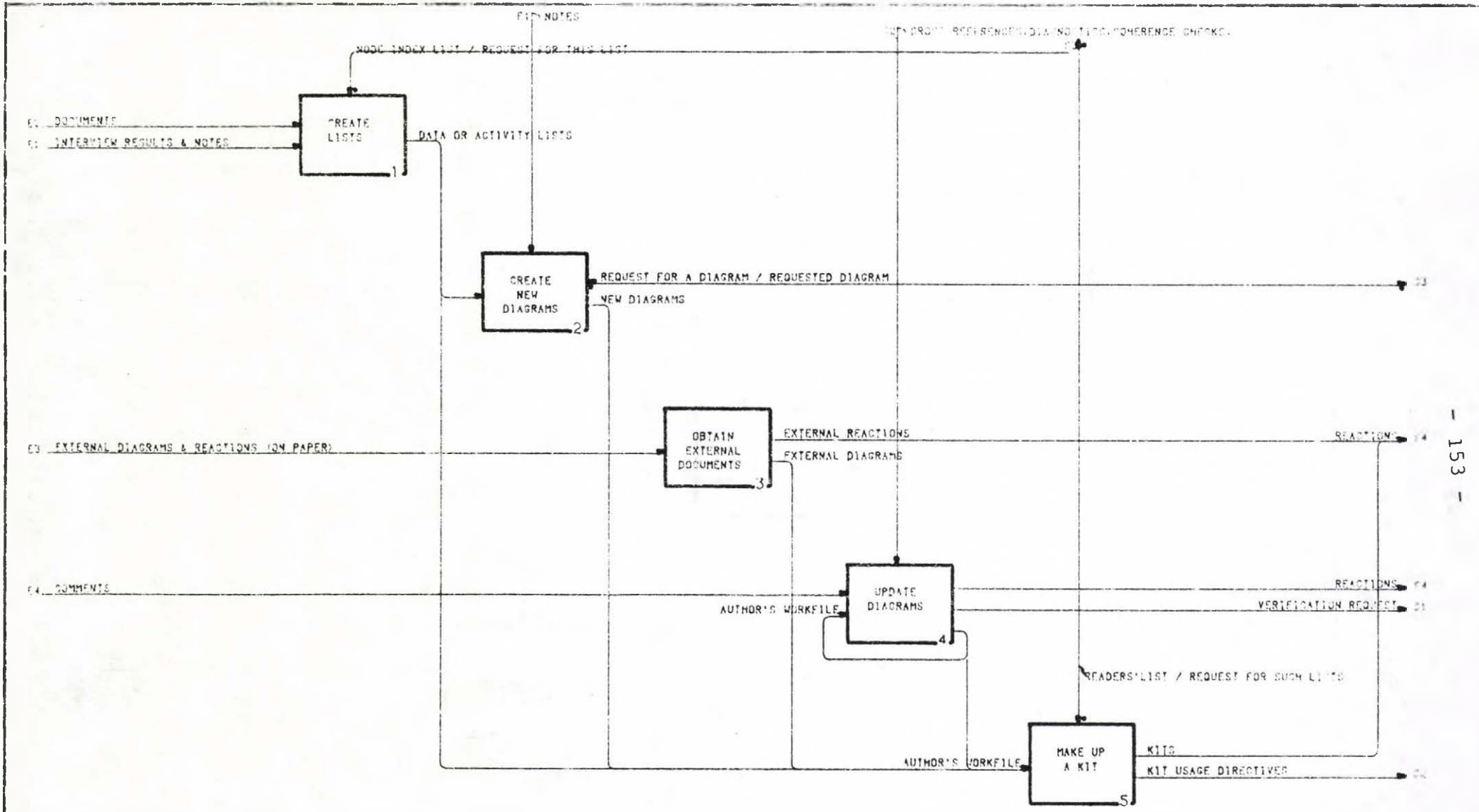
- Handle models (Diagram A0, box 4)
  (more specifically : Supply users with documents (Diagram A4, box 3)



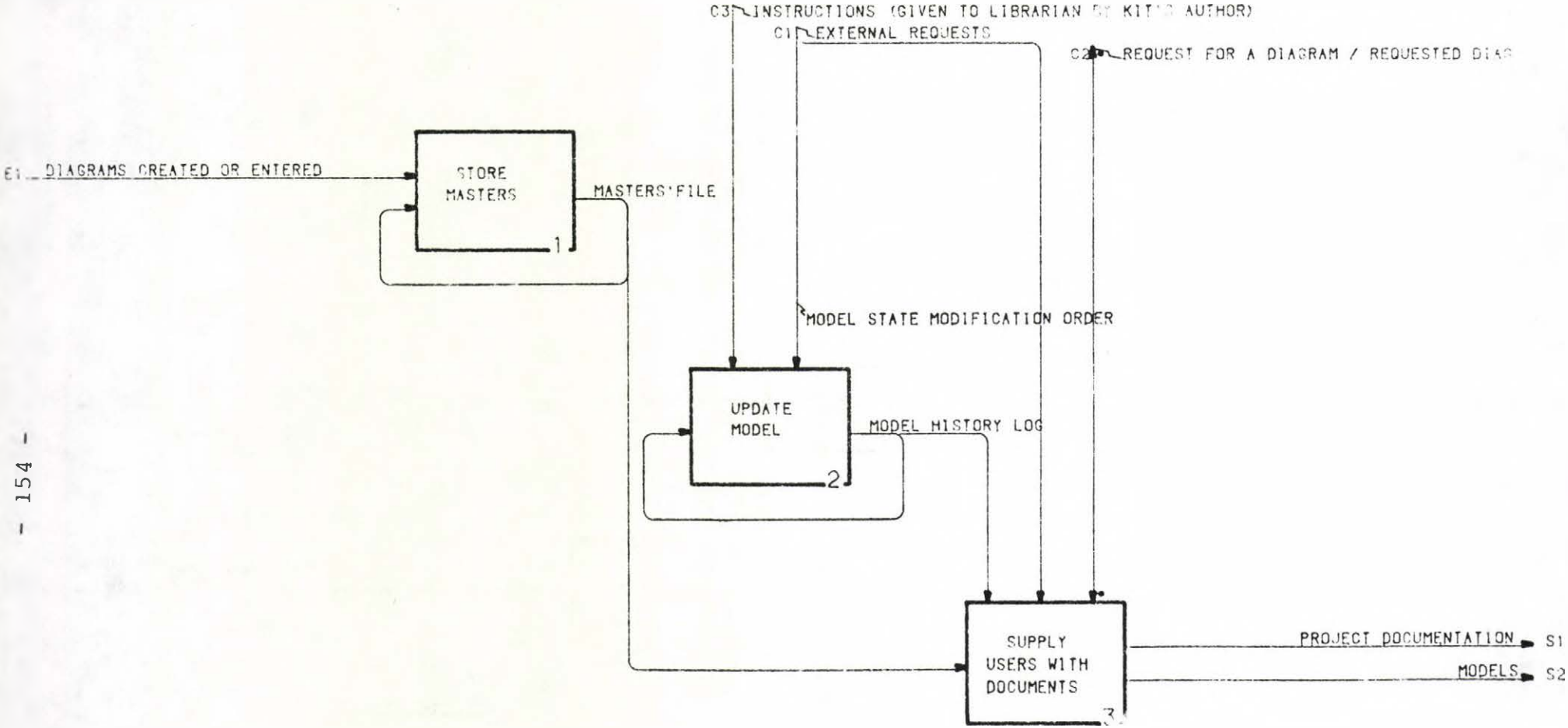
NOEUD : SAS/A-0	TITRE : ASSIST S.A.D.T. USERS	NUMERO :



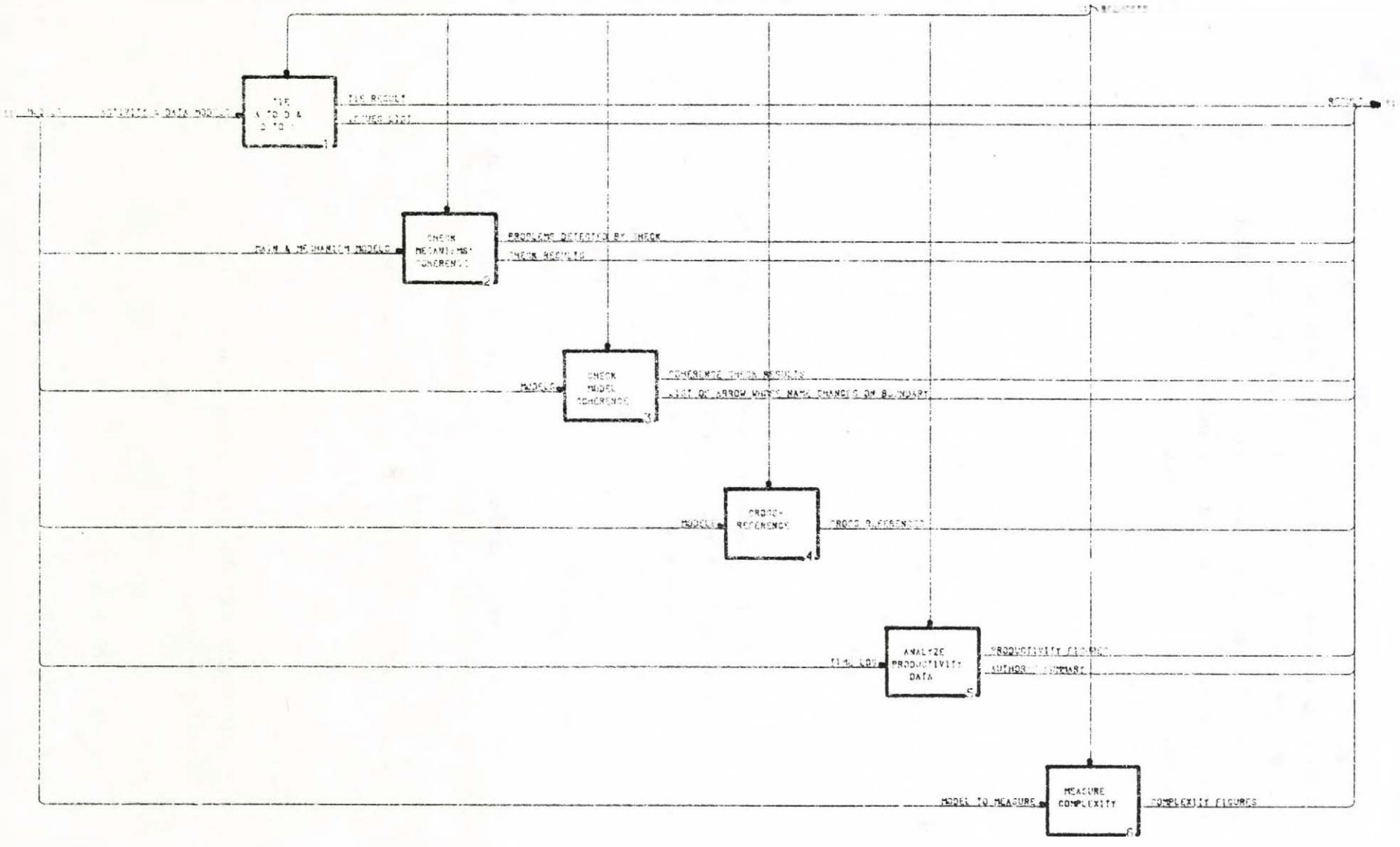
NOEUD : SAS/AO	TITRE : ASSIST S.A.D.T. USERS	NUMERO :



NOEUD : SAS/A1	TITRE : CREATE KITS	NUMERO :



MODELE : SAS/A4	TITRE : HANDLE MODELS	NUMERO :



NOEUD : SAS/A5	TITRE : MESURE AND VERIFY	NUMERO :

The legibility criterion

SADT syntax rules are precise but not very numerous. Therefore the SADT author has some freedom of action when drawing diagrams. Within the bounds set by the rules, he has to strain after the highest readability. To this end, he may set out boxes, arrows and labels "at best". The method gives guidelines for this drawing process, but the author's perception — conscious or unconscious — of the clarity of his diagram and of the understandability of his message also plays a large part.

SAS does not hinder the author from creating clear diagrams. To a certain extent, SAS enhances even diagram clarity, from the first step, when the first draft emerges from the previous sketches, to the last ones, when the draft evolves with the various comments and revisions. SAS has been designed to meet that purpose, whatever input/ouput device and operating mode is choses. This had led to establishing a trade-off between the number of data to input to SAS and the complexity of the algorithms providing automated drawing.

In fact, this trade-off point moves, while creating a diagram, fro a higher degree of automation, at the beginning to a lower one later, when ne author tries to enhance the legibility, once the correctness and the completude have been checked. This move is of course limited in batch input mode, but can be done iteratively, according to the author's "style" when creating a diagram element after element.

However this feature will depend upon the characteristics of the input/output devices which will be used with SAS.

These devices may have various graphic resolutions and various interaction capabilities. They may be choses among the following :

- Input devices :
  - graphic screen (700 K pixels minimum, with any pointing device : light pen, "mouse", cross-hair...)
  - digitalizer
  - semi-graphic screen
  - alphanumeric screen

♦ Output devices

- graphic screen
- plotter
- semi-graphic screen
- semi-graphic printer

SAS does not allow the output of diagrams on alphanumeric screens or printers, as the quality of such diagrams would be so poor that the advantages of SADT would be lost, for the most part.

Operation modes

As mentioned above, SAS can be used in a variety of ways.

♦ Interactive input : will be used in two major circumstances :

- Complete creation or modification of a diagram on a graphic work-station.

This creation will be done in any order, according to the actual sequence of thought of the author.

Any logical component of a diagram can be added, modified or suppressed to the being built diagram by performing the adequate sequence of steps among the following :

step a : selection of a basic component type (box, arrow, label, ICOM code).

step b : constitution of a complex component (e.g. : labelled box, arrow network, ...).

step c : selection of a component (basic or complex), already existing in a diagram.

step d : allocation of position parameters to a component (or modification of previously allocated ones).

step e : declustering of a complex component into its basic constituants.

All possible combinations of these steps are allowed : if step c may be sufficient to delete an item, one may need successively c, a, b, d to modify an existing component (while increasing its complexity).

As discussed below, step d is not mandatory : when enough information is provided to SAS, an automatic drawing is displayed. The author has then the possibility to modify e.g. the geometry of an arrow or the position of a label in order to increase the diagram's legibility.

– Transfer of a sketch (previously done with paper and pencil) into a clear diagram.

In this case, the author knows, from the very start of his/her work session, all the diagram's components and their relationships.

Consequently, the input of these components will be done by answering system's prompts and in batch input mode (see below).

When the whole diagram is entered (and automatically drawn), the author may, as he would have done in the previous case, modify the position parameters of the diagram's components.

This operation mode is preferably used on a graphic workstation, but remains possible with only an alphanumeric keyboard. However, in the latter case, the interaction is somewhat limited as the author will have to consider the plotter output to determine where space is available to modify the diagram accordingly.

Modifications of the contents of diagrams entered in the above mode are done as in the first case, by updating their elements, basic or complex.

◆ Batch input

In this mode, diagrams are previously drawn manually by a SADT author, then entered by an operator who will merely describe the existing diagram without trying to modify its layout.

The system will issue a set of prompts, designed such as to minimize both the risk of omissions and the numbers of keys to press. The dialog uses the SADT reference language (such as 1C2 meaning "second control of box 2) and follows the sequence below :

- number of boxes
- position (coarsely defined) of the boxes
- for each box :
  - label of the box
  - for each output arrow :
    .. label and attributes (two-ways ? tunneled ?) at origin
    .. for each destination :
    . label, attributes and destination identity
- for each external input or control arrow :
  - label, attributes and origin identity at origin
  - for each destination :
    .. label, attributes and destination identity

Answers and prompts will be displayed in tabular form : e.g. all data regarding a given arrow are displayed in the same screen.

Modifications of a diagram entered in batch mode can be made by consulting and updating the tables filled in while entering the diagram, or as if it had been entered in interactive mode.

♦ Syntactical checks

The SADT syntax is checked by SAS : first, while entering a diagram, to determine whether an element or an answer to a prompt is illegal (e.g. one cannot specify an arrow joining the input side of a box to the control side of another), second, when the diagram is declared as complete by its author or the operator, to determine whether the combination of elements is illegal (e.g. one cannot specify two arrows with the same label).

♦ Use & Output

Diagrams entered by one of the above means may then be :

- displayed on a graphic or semi-graphic screen ;
- hard-copied (plotter, screen printer, graphic printer) ;
- transferred between nodes of a local network if the input/output devices are not the same at each node ;
- syntactically checked within a model in order to verify that the parent-children relationship is correct (e.g. correctness of the ICOM codes) ;
- collected into a kit for further reading (this involves an update of the current model) ;
- enriched with comments (from readers) or with reactions (from authors).

Internal form

At this point of the discussion, it is worth noting the following : the diagrams presented in figures 3 to 7 are functional activity diagrams. They represent what functions SAS has to perform, not how SAS performs them. Consequently, the same word "diagram" (and "kit" or "model" which are made up by diagram assemblies) may represent a diagram (or a sketch of a diagram) on paper, a diagram as internally handled by SAS or the output, printed or displayed.

The internal form in which diagrams are manipulated by SAS is of primary importance : it is this form which will contain all data necessary to redraw a diagram, whatever the output device selected, but also to check its syntax, establish various cross-references (e.g. : list all activities controlled by a specific data, list all diagrams linked together by the "USED AT" fields,...) and perform some measurements.

3.2. Quality assurance functions

This aspect could well be the purpose of another model of SAS, but build with a different viewpoint. In the model shown, the quality assurance is shared out among box 2 and box 5 of the A0 diagram.

Reader-Author cycle

SAS entirely supports the reader-author cycle, main means to ensure quality when using SADT. This procedure is summarized in Figure 8.

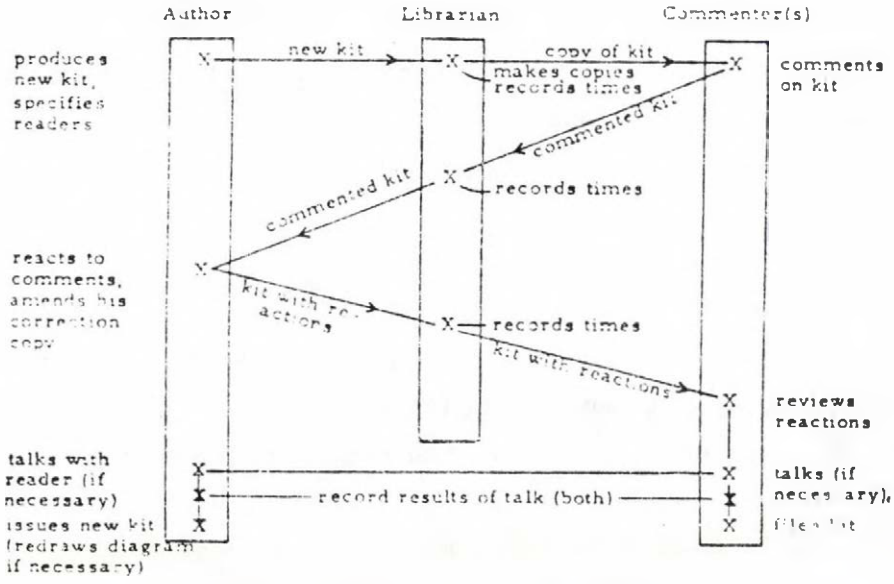


Figure 8 — The SADT Reader/Author Cycle

SAS ensures the team coordination by means of a mailbox which implements the communication and records the various activities performed. As an example, for every kit, SAS will record the kit number, the readers' list, the urgency level, the times at which each reader (or the author) has to comment upon the kit (or to react to the comments), and the actual times at which it has been done.

On SAS configurations allowing authors and readers to directly create, modify, comment and react at their workstations, all the librarian's functions will be performed by SAS. On limited configurations, or for users not willing to use a workstation, an operator will perform the librarian's copying and distributing jobs.

Measurements

As SAS handles diagrams in an internal form suited to algorithmic treatments, it becomes easy to obtain quality measurements. Structural complexity metrics allow SAS to rank any diagram from "Too simple" to "Too complex", thus permitting the project leader to tune the verification effort needed by the various portions of the model and possibly the development and quality assurance resources needed for the later phases of the project.

Productivity measurements are directly obtained from the records gathered at the mail box level and indirectly, from each author's workspace.

4. SAS STATUS

The development of SAS, which started two years ago is of the incremental type. That is, a nucleus has been first realized. This nucleus, called release 1, is a must for any environment or configuration. In addition to this nucleus, additional releases are being, or will be developed. In theory, as SAS configuration can be made up by adding any number of releases to release 1, but in fact, it is very likely that release 1, 2, 3 and may be 4 will be requested by most users in large companies, whereas small organizations may still benefit of a cheaper and simpler tool, made up by releases 1, 2 and 6.

Release 1

Uses currently a minimal configuration made up by a CALCOMP plotter and a VT 100 input device, running under VMS on a VAX 11/780. It performs the funtions represented by boxes 3 and 4 in diagram A2. It allows entering diagrams (only in batch input mode), storing them, and editing them, as long as the edition does not modify the number of the boxes. Syntactic controls are made at the level of the diagram only. The plotter performs the output.

Release 2 (Summer 1983)

Will use the same hardware configuration as release 1, but under UNIX * [UNIX-78] . It will perform the functions represented by boxes 2 and 3 in diagram A0. It will allow a hierarchical storage of all diagrams belonging to a model, coherence checks between diagrams, and will perform the basic duties of the librarian : handling diagrams, kits, models, and controlling the reader-author cycle.

Release 3 (Winter 1983)

Will use a hardware configuration using the VS100 graphic device. It will perform the functions represented by boxes 1 & 2 in diagram A1, allowing interactive creation (and modification) of diagrams, whatever the sequence in which the various diagram elements are given to the system by its user.

Release 4 (Spring 1984)

Will use a configuration where VT100's and VS100's are mixed. Instead of letting each user organize through UNIX the access to his files, it will manage all projects and author's files. It will completely perform the functions represented by boxes 2 & 3 in diagram A0.

Release 5 (Summer 1985)

Will use a database which will allow to perform the functions represented by the six boxes in diagram A5 and by box 1 in diagram A0. In addition, the functions represented by diagram A4 will be implemented differently, allowing to establish links with the activities performed later in the software life cycle.

Release 6 (Spring 1985)

Will allow the use of a semi-graphic printer as output device.

Note : Other tools providing extensions to the SADT method are also being investigated. We hope current research in this area will extend SAS to include such things as functional simulation, test generation, sequencing, etc.

---

5. CONCLUSION

SAS is a support tool which, if compared with other tools ([BARINA-79], [SMITH-81]), which also support SADT, has a set of characteristics making it particularly useful. Figure 9 summarizes these characteristics.

Tool	SAS	IMP	AUTOIDEF-0	CATHERINE	PASILA
Builder	IGL	Grumman	Softech & al.	Nokia, Softech	Triumph-Adler
Hardware	Independent	Independent	CYBER	PDP 11	TA
O.S.	UNIX	?	CYBERNET NOS	UNIX	?
Input devices	VT100 & VS100	Tektronix 4014/1015	Tektronix 4014/4015	Alphanumeric VDU	Alphanumeric VDU
Output devices	Plotter CALCOMP, Hard Copy	Plotter CALCOMP, Hard Copy	Plotter CALCOMP, Hard Copy	Graphic printer (DEC)	Graphic printer Plotter
Interactive creation & edition	Yes	Yes	Yes	Yes	Yes
Batch input	Yes	No	No	No	Yes
Coherence checks	Yes	Yes	Yes	No	Yes
Librarian functions	Yes	No	Yes	No	No
Quality Measurements	Yes	No	No	No	No

Figure 9 — Characteristics Summary

REFERENCES

BARINA-79 — Howard BARINA, W. COBEY, J. ROSENBAUM, Stéphanie WHITE
"Automated Software Design"
Proceedings of IEEE'S third COMPSAC, November 6-8, 1979.

CONNOR-80 — Michael F. CONNOR
"Structured Analysis and Design Technique — SADT"
Portfolio 32.04.02 — System Development Management
Pennsauken NJ : Auerbach Publishers, 1980.

DICKOVER-77 — Melvin E. DICKOVER, Clement L. Mc GOWAN, Douglas T. ROSS
"Software Design Using SADT"
Proceedings of the 1977 Annual Conference of the ACM,
Seattle, Washington, October 16-19, 1977, pp. 125-133.

MYERS-75 — Glenford J. MYERS
"Related Software through Composite Design"
New York, NY : Van Nostrand Reinhold Co, 1975, p. 30.

ROSS-76 — Douglas T. ROSS, John W. BRACKETT
"An Approach to Structured Analysis"
Computer Decisions (9), Sept. 1976, pp. 40-44.

ROSS-77a — Douglas T. ROSS, Kenneth E. SCHOMAN Jr
"Structured Analysis for Requirements Definition"
IEEE Transactions on Sofware Engineering 3(1), Jan. 1977, pp. 6-15.

ROSS-77b — Douglas T. ROSS
"Structured Analysis (SA) : A Language for Communicating Ideas"
IEEE Transactions on Software Engineering 3(1), Jan. 1977, pp. 16-34.

SMITH-81 — Daniel G. SMITH
"AUTOIDEF-0 : A New Tool for Function Modeling"
Softech Document Number TP 125 - Sept. 1981.

UNIX-78 — (Several authors)
Special Issue on UNIX Time-Sharing System
BELL System Technical Journal 57(6), July-Aug. 1978, pp. 1897-2313.

WEINBERG-75 — Gerald M. WEINBERG
"An Introduction to General Systems Thinking"
New York, NY : John Wiley & Sons, 1975, p. 28.

THE USE OF PETRI NETS IN REQUIREMENTS AND FUNCTIONAL SPECIFICATION

M. Maiocchi - Istituto di Cibernetica di Milano
Etnoteam S.p.A. - Milano

1. Introduction

In the recent years different approaches have been carried on with the purpose of reducing the cost and increasing the quality of the software products; these methods are generally related to operating rules which guide the programmers in obtaining standard programs; often automatic tools are provided, both for cost reduction purposes and as a constraint in avoiding deviations from the rules. While a certain number of programming methodologies has been set up, which are currently widely used, the number of analysis methodologies is not so high, and more and more for requirements definition methods. In particular, we can see an increasing gap between the system programmer and the applicative programmer know how, in particular for business oriented programs: this fact is probably due to the different production environment of the two kinds of people, the former being close to software engineers, the latter to customers which are not skilled in computers. More, while the system programmers are surrounded by an organizational frame due to the tradition of this kind of development, the business oriented programmers have to deal with fast work group set up, no tradition, no pre-existent development organization.
The paper will present a method based on the use of the Petri Nets for driving the requirements and functional specification, and for allowing a reliable communication with the customer in order to avoid ambiguities on the purposes and on the functions of the system to be developed.

The method has been largely experienced: since 1977 more than ten projects have been successfully carried out (for which is known by the author: eight of them have been reviewed directly by the author), sizing about 3-7 man years each.

The method is located in a software production cycle which is shown in fig. 1:

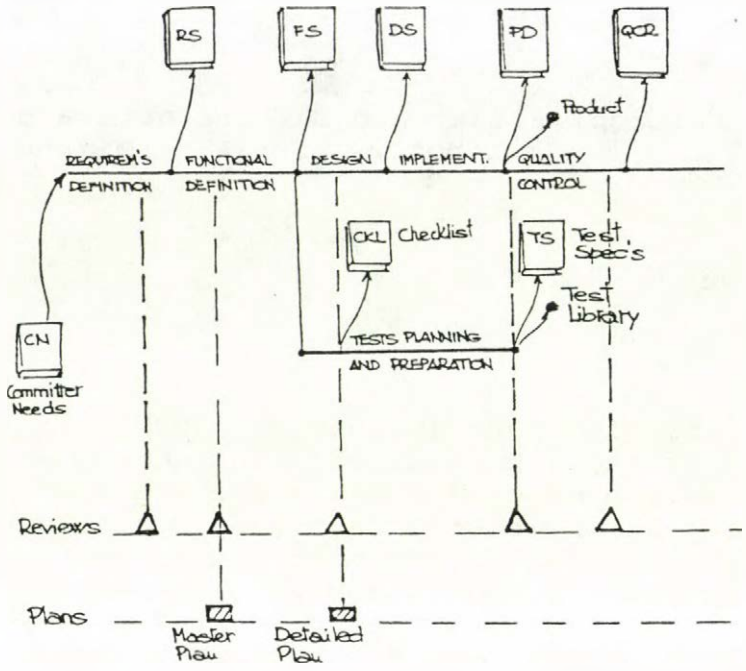


Fig. 1

- starting from a document containing the Committer Needs, a phase of Requirements Definition produces a document of Requirements Specification, which is input to the
- phase of Functions Definition, which produces a document of Functional Specification containing also a high level description of the architecture of the product;
- following the FS document, it is possible to perform the Design phase, which provides the detailed Design Specification, followed by
- the Implementation, which provides the sw product andthe document Product Description, which is a refinement and completion of the DS, which takes into account the implementation details;
- concurrently with the Design and the Implementation phases, the Test Planning and Preparation is performed, which produces, starting from the FS, the Checklists (list of the items to be controlled), the Tests Specification (which can be considered as the FS for the tests) and the Tests Library (that is the programs and the data to be used for controlling the product);
- after the completion of the tests and of the product, the Quality Control phase produces a Quality Control Report, through a controlled tests execution, referring about the quantity of the testing performed and about the resulting measured quality of the product.

After the release of the first version of the product, each of its parts (code, source, documents, etc.) is "frozen" and the maintenance or the continuation follows repeating the above phases on physically different pieces: a specific maintenance phase is not recognized.

Documentation, Planning and Review are other activities always present during each phase, but which cannot be assumed as specific phases.

In the following paragraphs we will present and discuss the various methods set up, integrated and experienced in each activity.

2. PURPOSES OF THE METHOD

2.1 Requirements definition

We consider "requirements" of a software system the desired behavior of the complete human and machine environment in which the product will run.

For this reason, we need the capability of describing in a simple, unambiguous way all the human and automatic procedures we want in a specific environment.

Such procedures can be characterized by:

. sequentiality or concurrency in the time;
. causal dependency or independency;
. starting of activities connected to the presence of condition status, resources;
. production, occupation, consumption of resources;
. production of conditions, status.

2.2 Functions definition

The activity of functions definition has the purpose of providing a complete indication of the functions supported by the product, and of the "languages" for communicating with it:

. launch procedures
. commands
. input data and relative syntax
. ouput results and relative syntax
. error messages
. video masks
. etc.

Furthermore, the high level architecture of the product must be defined, in order to provide any information suitable for constructing development plans, and for evaluating the cost for the implementation. By 'high level architecture' we mean the specification of the main building modules of the product, of their functional roles within the product, and a complete specification of the interfaces between the modules themselves and with the host system.

No details are given on the internal subcomponents: these details are given in the Design Specification.

2.3 Customer/supplier Communication

One of the main problems in the definition of business oriented systems arises from the ambiguity in the communi cation betwen the customer requiring the system and the supplier which must build up it; the committer is general ly unskilled in computers, does'nt know exactly what he want, is not conscious of the problems (and the cost) due to changes in specifications during the development, under-evaluates the organizational role of a computer and the difficulties in changing his own organization; the committee is never shure that the system described in the proposed documents has been deeply and correctly examined and understood.

The resulting development activity suffers of changes in the specifications, of discoveries of new items to be examined, of unsactisfaction in performances and in the ease of use of the final product: that is high costs and low quality.

The method based on the Petri nets addresses to "sharpen" the communication between committer and committee, making it reliable, unambiguous and allowing early reviews.

2.4 The development groups

For historical reason, the programmers devoted to business application use mainly the programming Language COBOL, and are educated through professional courses, avoiding accura tely all the basic knowledges of the computer science (they don't know at all terms as "predicates calculus", "recursion" "concurrency" and so on); this fact can produce rejection of new methods which could appear as too formal, too abstract, too impositive; the change of this condition cannot be provided within a project or with some kind of training: it is required a cultural growth which imposes long periods; the method based on the Petri nets is particularly suitable for the environment depicted above, because:

- it has a "gentle face", which describes the object the pro grammer deal to, in terms of the real objects, but allow slippering toward abstraction;
- it can be used partially, obtaining advantages proportioned to the partial use (the completely formal methods requires the complete carrying out of the method for obtaing the re sult: a partial application gives generally no results at all).

3. THE METHOD

3.1 Petri Nets

A Petri net is a "bipartite" graph in which two kinds of nodes are recognizable: places and transitions. Oriented arcs connect the two kinds of nodes, so that no two places are connected together and no two transitions are connected together. Fig. 2 represents a Petri net in which two transitions occur, on with two places in input and with one place in output, one with one place in input (shared with the previous transition) and with two places in output.
The Petri nets can be interpreted as the description of a process in which events can occur depending on a set of conditions, determining other conditions: in particular, each place can be interpreted as a resource and each transition can be interpreted as an activity: each activity can take place only when their input resources are present, and its happening produces the output resorces, consuming the input ones (Fig. 3).
By the way, the Petri nets can be used in representing production processes, and in particular, programs connected together in a procedure.
The presence of a resource in a net is represented through a mark in a place (indicated with a dot) (Fig. 4) and the temporal evolution of a net can be represented through the flowing of the dots through the net itself: when an activity "fires" the marks are drawn from the input and a mark is put on each output place. For example, the net of Fig. 4 represent a process in a medical analysis laboratory: when the receptions desk is free and an applicant is present, the acceptance

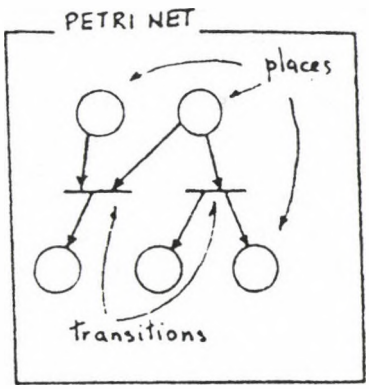


Fig. 2

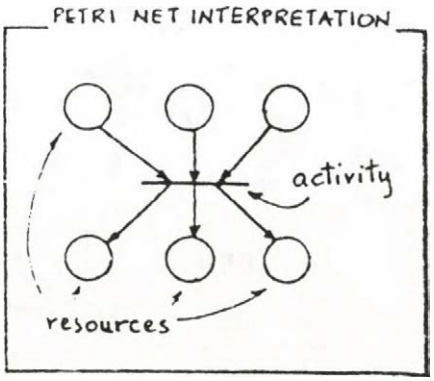


Fig. 3

operation can take place, producing a test request form which, together with the presence of a doctor and of the patient, can fire the drawing, producing a sample and an updating of the test request form; the reporting activity must wait for the production of the result through the analysis activity, producing then the final report.

3.2 Petri Nets Graphical Modification

In order to make more apt the Petri nets to our purposes, two kinds of graphical changes have been introduced: the former, an highly evocative representation; the latter, a set of summarizing forms for common situations.
In Fig. 5 is represented an evocative net, in which, each resource has been replaced with a graphical

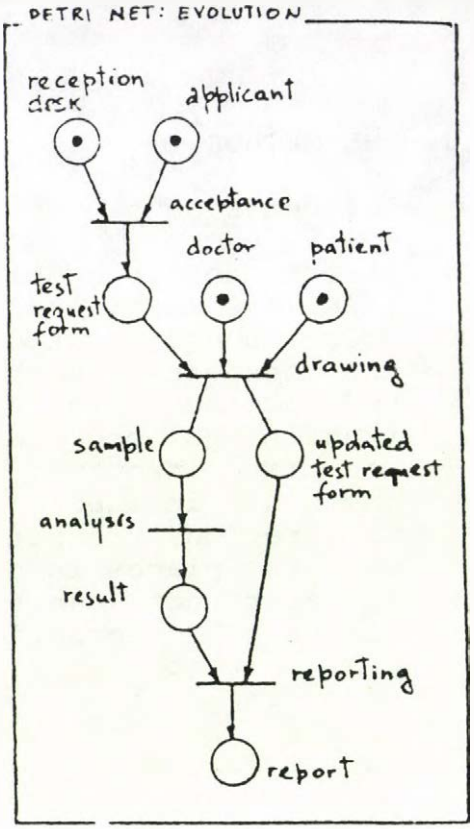


Fig. 4

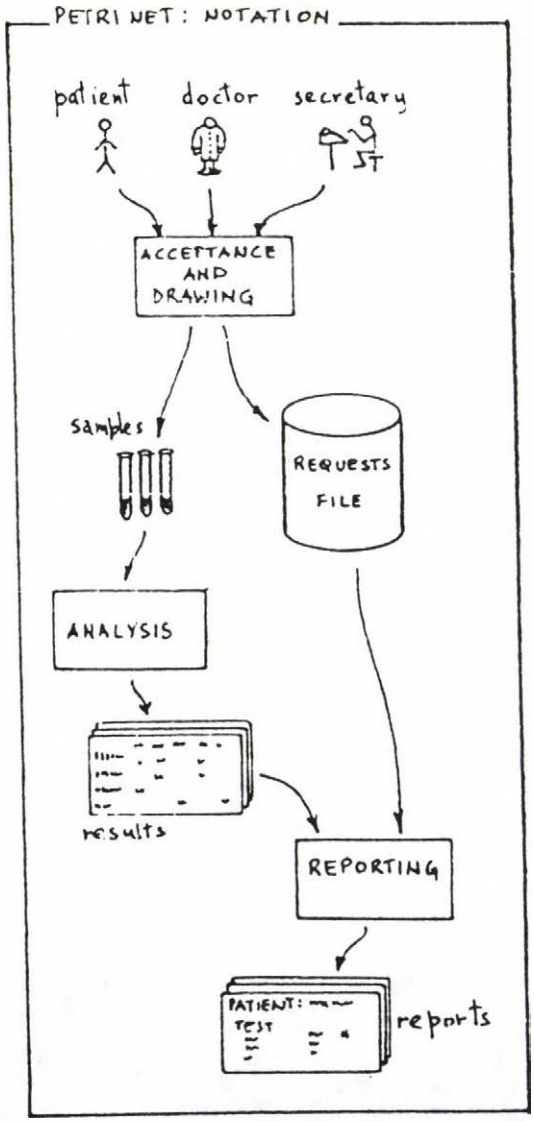


Fig. 5

symbol representing the problem entity in a mnemonic form, and each transition has been represented as a box containing the description of the performed transformation: the net represents more concisely the work of the previous medical analysis laboratory, in wich acceptance and drawing phases have been put together.
The second kind of modification refers to the capability of representing parallel or alternate feeding of the transitions or generation of the resources: for example,

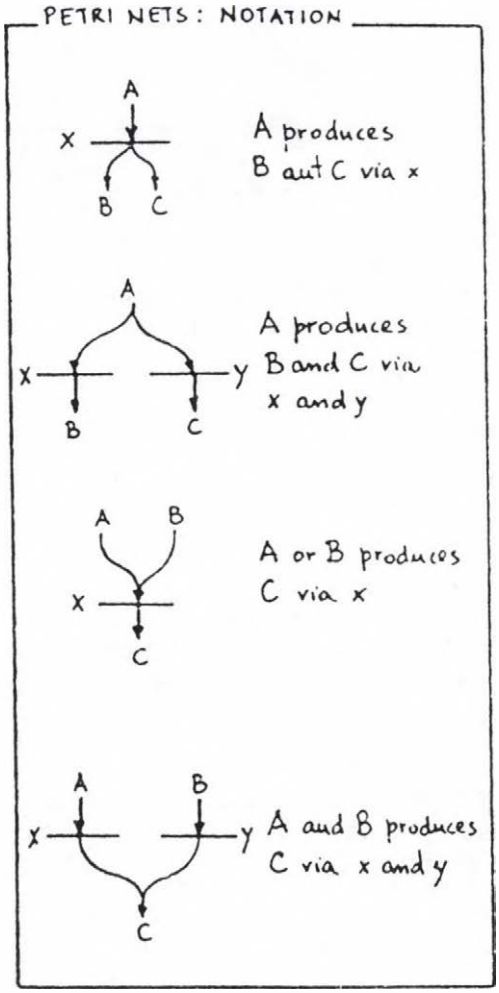


Fig. 6

the first form of Fig. 6 is useful for representing the operations on a file A which can alternatively produce a file B or an error message C: the second form can represent two programs x and y accessing simultaneously a file A; and so on.
The illustrated extensions of the graphical form are not extensions of the Petri nets capabilities: in fact each new form can be exploded into traditional Petri nets, in which,nevertheless, it is difficult to attach a meaning in term of problem entities to each net element; for instance, Fig. 7

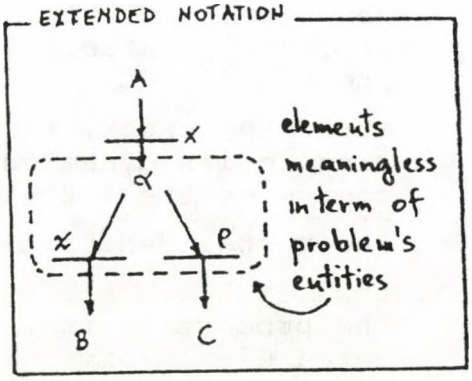


Fig. 7

shows the constructions of the first form of Fig. 6 through traditional Petri nets: no meaning can be connected to the resource and to the transitions χ and ρ.
The Fig. 8 shows another extension, in which it is possible intuively to recognize that the resorces, through the transition x, one of the two resources D or the couple B and C.

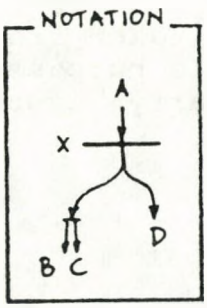


Fig. 8

3.3 The method

The method is summarized in Fig. 9, through a Petri net:

0. a first step defines the complete user environment in which the product must be inserted as a single Petri net, constituted by a single transition, with evidence to the resources or to the conditions significant as input and output of the whole activity;

2. the net is then analized following a table called of "local checks" (see later), which sets questions to the analyst about the kinds of connections between the transition and the resources, in order to verify the adequacy of the net to the intentions of the analyst itself;

3. then the net is joined to a verbal description, in which the significant attributes of the resources are described (e.g., in a driven way, the accesses to the files, the form of the commands, the kind and the number of some sheets, the skill of the involved people, etc.), and the performed activity is defined (what is performed, not how);

4. in the following step, a linguistic check is carried on, in which all the parts of the language are examined in order to avoid ambiguities (unnecessary attributes, imprecise articles, undefined numbers, impersonal forms, etc.);

5. then the net is exploded in another one, in which more than a transition is defined (generally up to 6 - 7);

6. the exploded net is controlled through the so called "contextual checks" table (see later), which sets questions about the possible ways in which activities can occur (concurrently, interleaved, sequential, mutually exclusive, etc.);

7. after the check, the net is anew verbally described, referring to its dynamical behavior, whithout regard on the resource or the activities;

8. the verbal description is then checked;

1. then each transition is insulated from the context, and the process is iterated until the transitions can be considered elementary , from the point of view of the executor (that is, when a transition is operated by the some human or automatic interpreter, or by a complex not involved in the area to be changed by the product).

Let we see the complete cycle on a small example (taken from a real product and extremely simplified for example purposes), for the functional specification of a "cash and carry system.

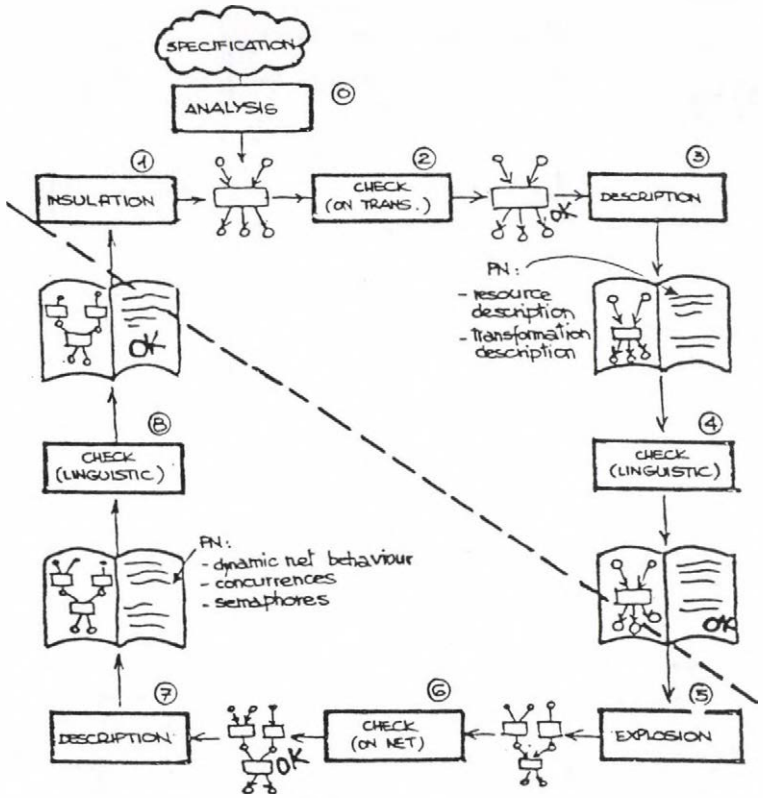


Fig. 9

3.4 An example

3.4.0 Top level procedure description.

In this step the complete procedure is represented as a unique transition, and the resources involved are indicated in a concise way, collecting together the resources with some similarity of use or of nature (for instance, all the disk files are indicated as an unique resource). Fig. 10 shows an example of a system of terminals cash and carry which can operate concurrently performing invoicing operations: the complete procedure is seen as a unique transition, which has as input resources:

- the unactive system, which will be activated through
- initialization commands;
- execution commands, which allow the invoicing operations from the single different terminals, accessing a set of
- files, containing informations on the customers, on the products, etc.

The output resources are:

. the printed invoices;

. the terminal system, which will be returned in a unactive state;

. the files, which will be released after the operations, in an update condition (the symbol '+' associated to the arc indicates an updating operation).

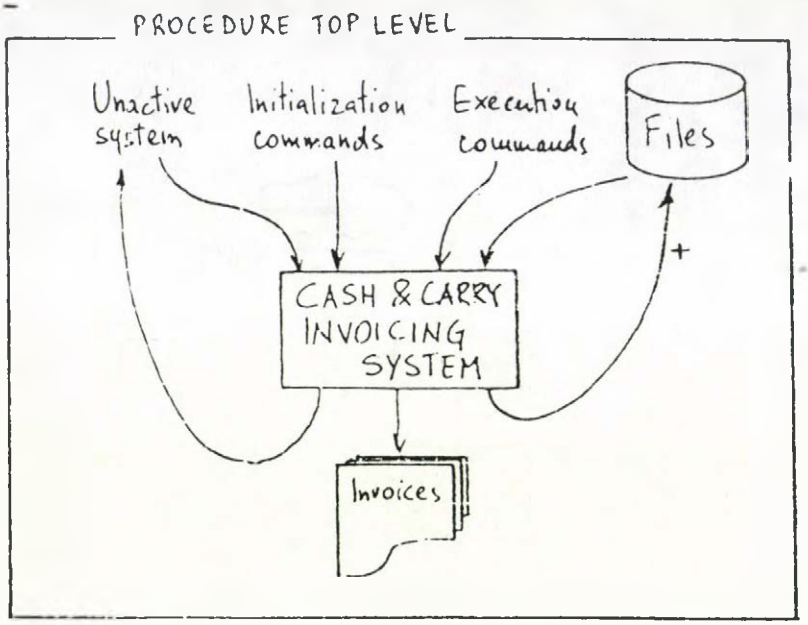


Fig. 10

3.4.2 Local check

The table shown in Fig. 11 collects each possible situation in respect of a single transition and the relative input or output resources; by this table we can examine the correctness of the net of Fig. 10:

The invoicing system (seen as a big black box) requires joined feeding of the unactive system, of initialization commands, of execution commands and of the proper files (until now not yet completely specified) and produces parallely the possibly up dated files, the invoices and the unactive system (after the daily operation completion).
It seems to be correct, so that we can carry on the activity.

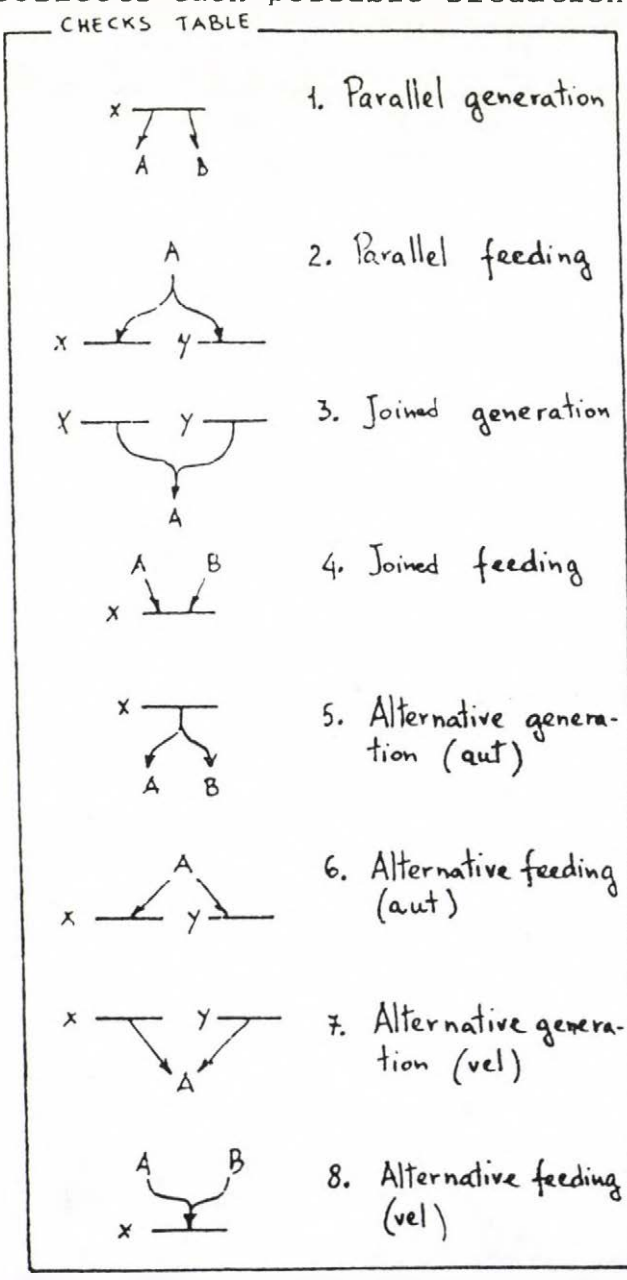


Fig. 11

3.4.3 Verbal description

The step requires the construction of a descriptive page for the completion of a PSPN module (°), in which:

. each resource is described in terms of its components and of their meaning in terms of problem entities;

. the transformation of the transition is specified in a summarized form.

So, we will define here the characteristics of the system, of the file (at this point, a rough classification will be adeguate), of the generic purposes of the commands, and so on, and the functions to be performed.

The main purpose of the description is to define concisely but in a not ambiguous way both the resources and the transformations of the net; therefore, we require that each resource is described in respect of:

. the kind of the physical support involved (disk, tape, keyboard, etc.);

. the physical characteristics of the resource (when disk: indexed, sequential, key format, size, ect.);

. the syntactical form (allowed characters and information sequences, when from keyboard; record layout when on disk, etc.);

. the synchronization needs (generally on disk resources) and the relative locking level (volume, physical record, or intermediate logical levels);

. the characteristics of the plural names, that is elements number (defined or undefined, and, if defined, maximum and minimum) and the predicate which define the set to which the elements belong.

The description of the transformation can be performed (when the cultural environment allows it) by means of a language

---

(°) The PSPN documentation technique requires the construction of the documents as a set of PSPN modules, each of them constituted by a couple of pages, in which the left one contains highly summarizing schemata, drawings or slogans, and the right one contains an indented verbal detailed explanation.

which mixes natural language to the construct shown in Fig. 12, in a fashion, as possible non-procedural: the constructs refer to:

. a set of operations to be performed, whithout specifying their sequèntiation;

. sequences of operations (sometimes it is the best way to specify transformations);

. conditional operations (when connected with the first construct they are quite similar to the "guarded commands" of Dijkstra);

. operations on "plurals".

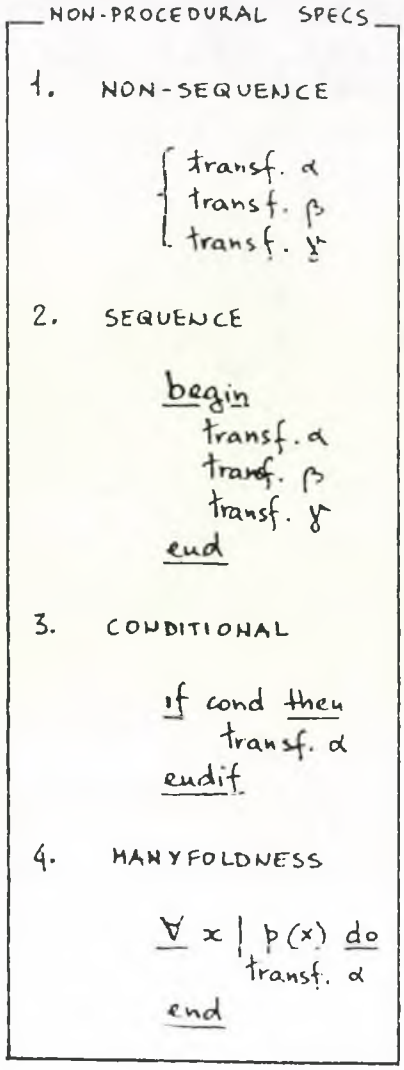


Fig. 12

3.4.4 Linguistic checks

The right part of a PSPN module, that is the linguistic description of the net, must be controlled too: the controls can be considered as very tedious, but they reveal very useful in avoiding the specifications ambiguity; they are referred to the different parts of the speech:

. article: we must verify and must be able to state the reason for their absence, definiteness or indefiniteness;

. noun: each plural or collective must be specified by a set of attributes for the individuation of the set to which the intended elements belong;

. adjective/adverb: they must be unavoidable;

. verb: subject and (for transitive verbs) object must be always expressed or in any case not ambiguous;

. pronoun: their reference must be clear;

. conjunction: they must be properly used; in particular, the or conjunction must be specified for representing an "aut" or a "vel" conjunction.

3.4.5 Petri net refinement

The next step consist of the refinement of the previous net, exploding the initial unique transition into a net in which more transitions appear, corresponding to different activity phases: the example of Fig. 10 will be developed as shown in Fig. 13, in which three different activities are individuated:

. the first one is the initialization of the system, which allows to obtain active terminals, able to accept commands;

. the second one is the reconfiguration of the system, allowing the activation of new terminals, or the deactivation of active terminals;

. the third one is the operation with the active terminals, for the invoicing purposes.

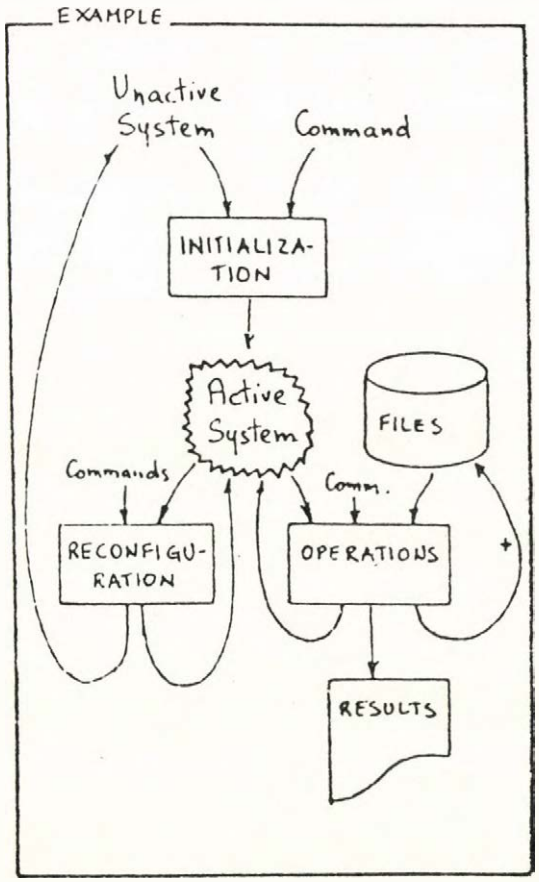


Fig. 13

3.4.6 Verification of the correctness of the net

The net describes completely the synchronization among the various activities of the procedure and we are then able to check the correctness of the description in respect of the problem requirements. In particular, the net imposes a precise description of the above synchronization aspects, so that possible lacks in the requirements will be recognized in the net as behaviour choices which must be analyzed for approvation.
The control activity is carried on through both the check table n.1 and the table of Fig. 14, in which more complete topological situations are catalogued.

For the local checks:

- the initialization requires the system unactive and some initialization command (correct), and generates the active system (correct);
- the reconfiguration requires the active system and some reconfiguration command provided through terminal (correct), but cannot produce simoultaneously the active and the unactive system as described: the net must be changed by introducing an alternative generation; the encountered error is evidently due to a mistake done by the designer, and no lacks in requirements can be observed at this point;
- the operations require the active system and the files and some operation command provided through the terminals for obtaining the correct invoices (correct), and generate the results (the printed invoices), updating the files and releasing the active system (correct);

CHECKS TABLE N. 2

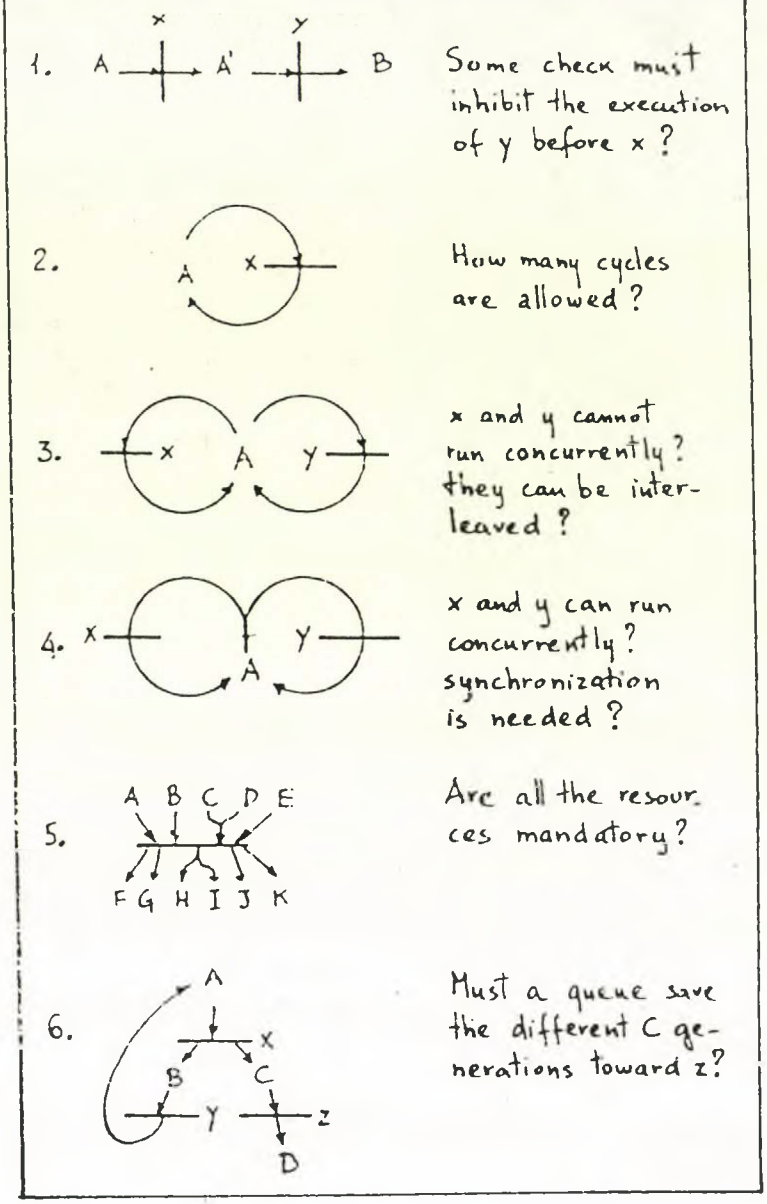


Fig. 14

- the active system can feed alternatively the reconfiguration activity or the operations activity, as shown in the net ; we cannot decide when the description is or not correct about this point, but a choice has been taken in the net drawing: a lack in the requirements made it possible; in fact , it is not acceptable that the activation of a new terminal requires the stop of the operation on other terminals: the net must be changed with a parallel feeding situation.

For the contextual checks:

referring to the previous example, as updated in Fig. 15 following the indications of the performed controls, we must verify and declare:

- limits in iterating the operations activity (in the case, none);
- limits in iterating the reconfiguration activity (in the case, none);
- capability of accessing concurrently the active system (in this case, the capability of turning off a terminal currently used, and some consequent protection about it);
- optionality of the resources: all are mandatory.

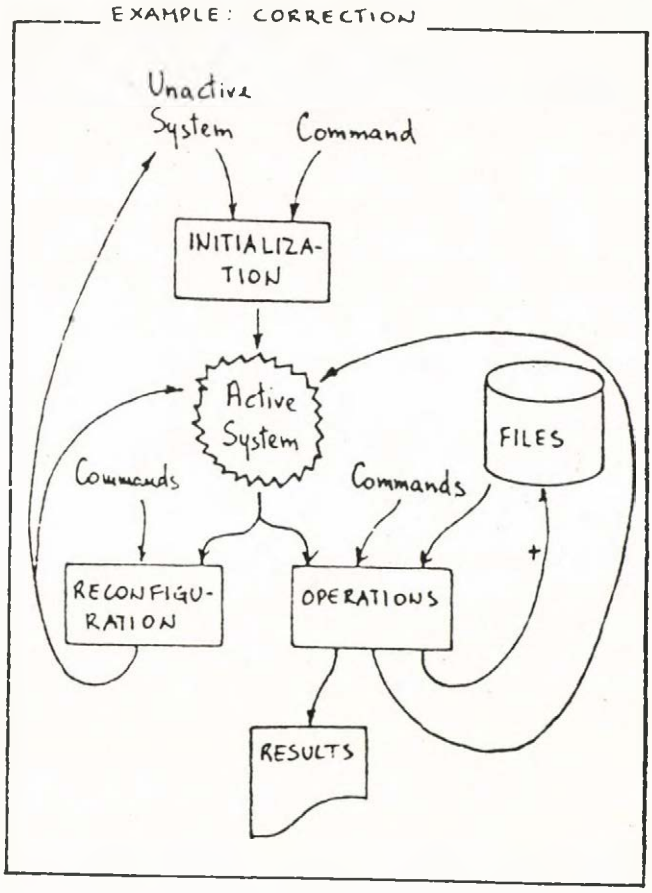


Fig. 15

Another not yet described check to be performed refers to the net complexity:
it is very hard to keep under control a net in which many resources and transitions appear, so that it is very important that the net explosion gives a small net at each step; this can be generally obtained through a reduced explosion of the resources (in the example, the unique splitted resource is the system, as "active" or "unactive"): for example, in the further developments, it will be necessary to distinguish the various involved files; probably, it will be useful to divide them first as anagraphic-read-only files and updated-work-files, and then, after other refinements, to individuate the specific resources such as the anagraphic-customer-file, the invoices-work-file, etc.

3.4.7 Net description

The net is described in the PSPN form, as performed in the step 3, showing the dynamic behavior, without defining the details on the resources or on the single transitions, which will be treated later.

3.4.8 Linguistic check

The activity is carried on as in the step 4.

3.4.1 Insulation

The specification of the procedure continues insulating each transition of the previous refinement together with the needed resources, cutting any not necessary arc. For example, the transition operations of Fig. 15 can be insulated and then refined as in Fig. 16 , in which two kinds of activities are recognizable.
The refinement is checked and completed with a linguistic description as described in the above steps.
(Note, in the example, that the resource files must be accessed concurrently and must be then semaphorized; because each invoice will occupy more than one record of an indexed invoice-file, it will be needed some semaphorization mechanism at the level of the invoice number, for avoiding concurrent access on the same invoice).

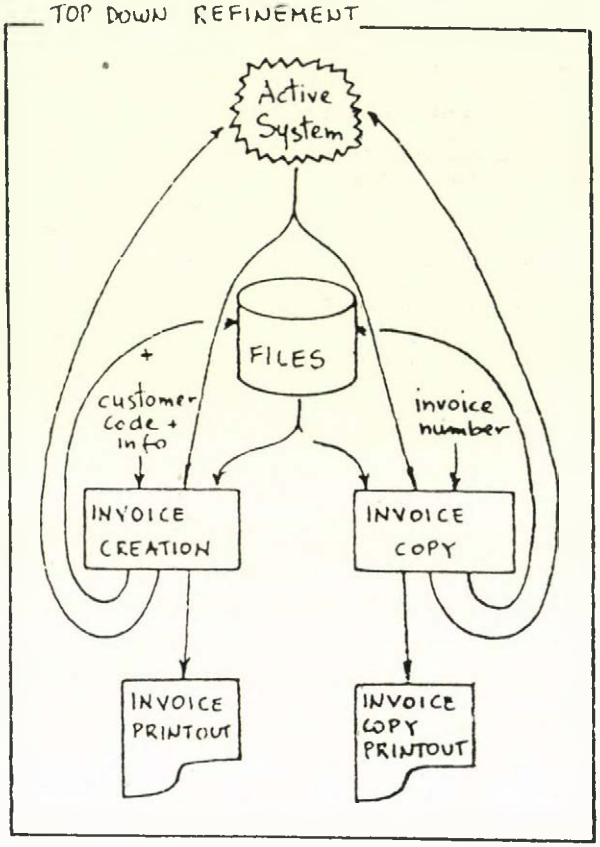


Fig. 16

The refinements go on until the program level is reached, that is until the transitions defined refer to a single program.
There is no way to define in a precise fashion what we mean as "program", because no limits can be effectively imposed to the top down development; nevertheless we can try to give our "definition" of "program level" and of "procedure":

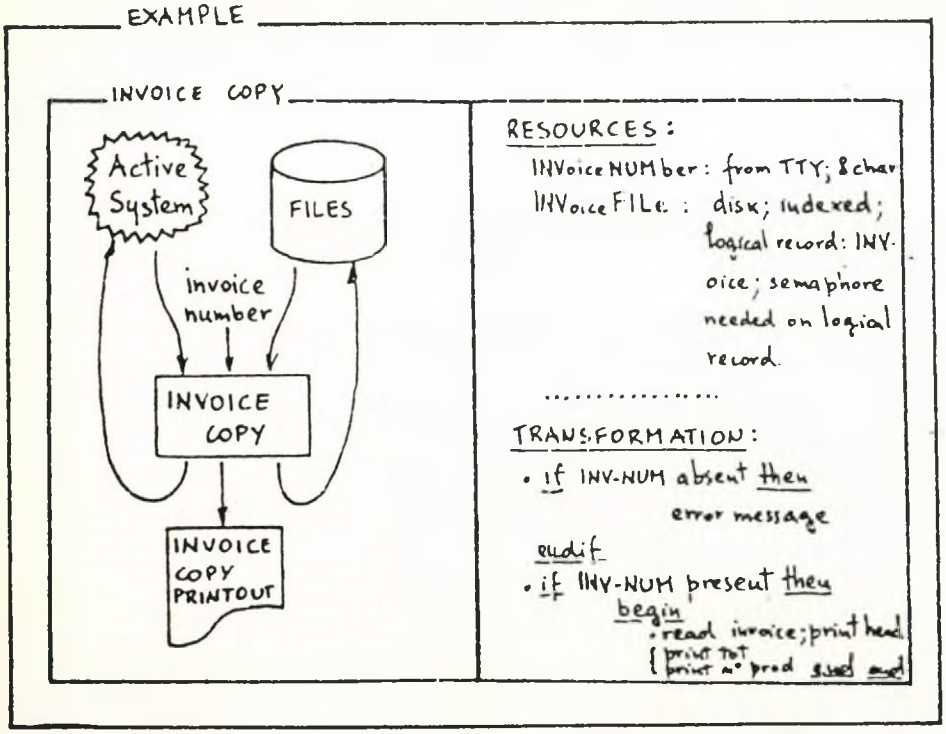


Fig. 17

a procedure is composed of a set of programs for which the control sequence (or the set of control sequences) cannot be completely fixed: we can think to different executors (machines or men) working concurrently or with some synchronization; a program is a set of operations for which the execution sequences can be completely deterministic.
The definitions get a reason by the need of different design methods: while programming methodologies such as Jackson's, Warnier's or PHOS can be applied to programs (in the sense expressed above), the shown methodology can cover also the upper levels of the procedures.

Fig. 17 shows a rough example for a program specification.

3.5 Operation timing and throughput of the whole system

The definition of the requirements of a system can be carried out exactly following the above method, inserting in the description all the involved activities, including the human ones.

We will show a sketch of a requirement description for a warehouse management, illustrating the capability of the method of forecasting the general throughput of the system, with suggestions for alternative organization.

Fig. 18 shows a possible organization:

the ware reception activity starts with a man devoted to this work, the arriving ware and an order list; the man ckecks the arrived ware on the list, classifying it into three classes: the ordered but not arrived is prepared for new ordering (to be noted that scarce details are provided on this activity,

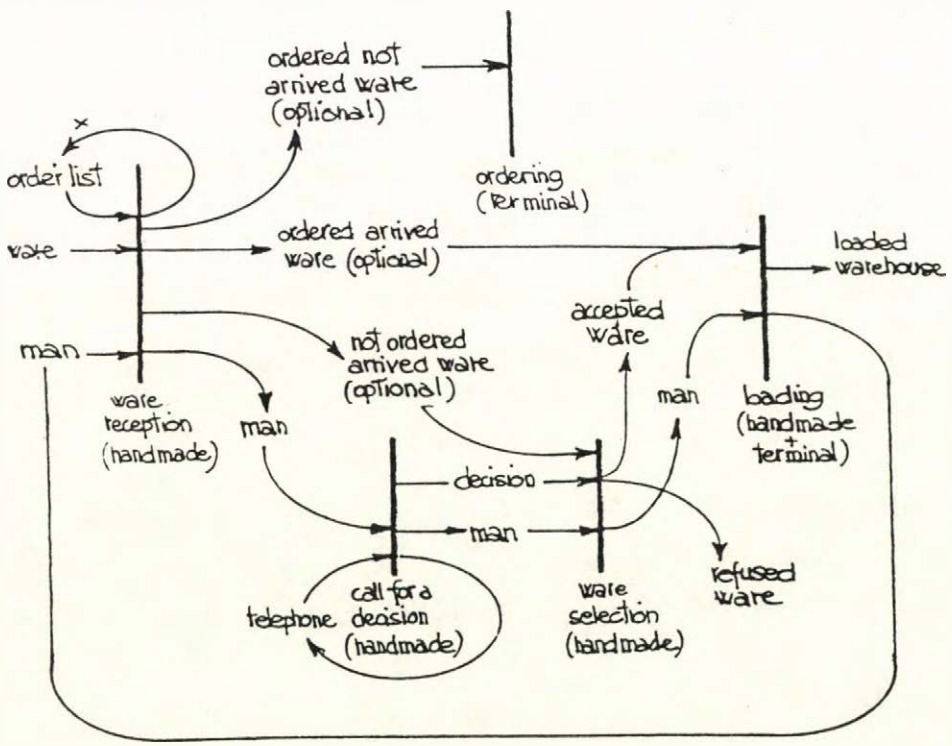


Fig. 18

not a goal of the current description); the ordered and arrived ware is input to the loading activity (which can not start for the absence of the man); the unordered arrived ware produces a call for a decision to the proper employee; the decision allows the ware selection for refusing a part of the ware, accepting a part and providing the resource "man" for the loading activity; at this point the man is anew available for the reception.

To be noted the direct connection providing the man to the loading when no decision are requested.

The complete net throughput can be computed as shown the following table:

ACTIVITY	MEAN HUMAN TIME SPENT	% OF THE OCCURRENCE	WEIGHTED TIME
Reception	45'	100	45'
Call for a decision	45' + 5'	15	≃ 12'
Ware selection	30'	15	≃ 5'
Loading	2h	100	2h
TOTAL TIME			≃ 3h

The time for call for a decision has been evaluated supposing 5 minutes for calling the responsable and 45'for obtaining decisions on the whole arrived ware,weighting the percentage of the occurrences as the 15% of unordered ware; a larger fix time for reaching a far telephone would evidentiate logistic problems.

At this point we can estimate the frequence of the invocation of the whole activity, for planning the proper number of employees: f.i., 2 arrivals in a day requires about 6 hours, that is 1 man , but the last arrival must occur not later than in the early afternoon; 3 men can operate simultaneously allowing 6 arrivals in a day.

A different organization can be throught, as shown in Fig.19

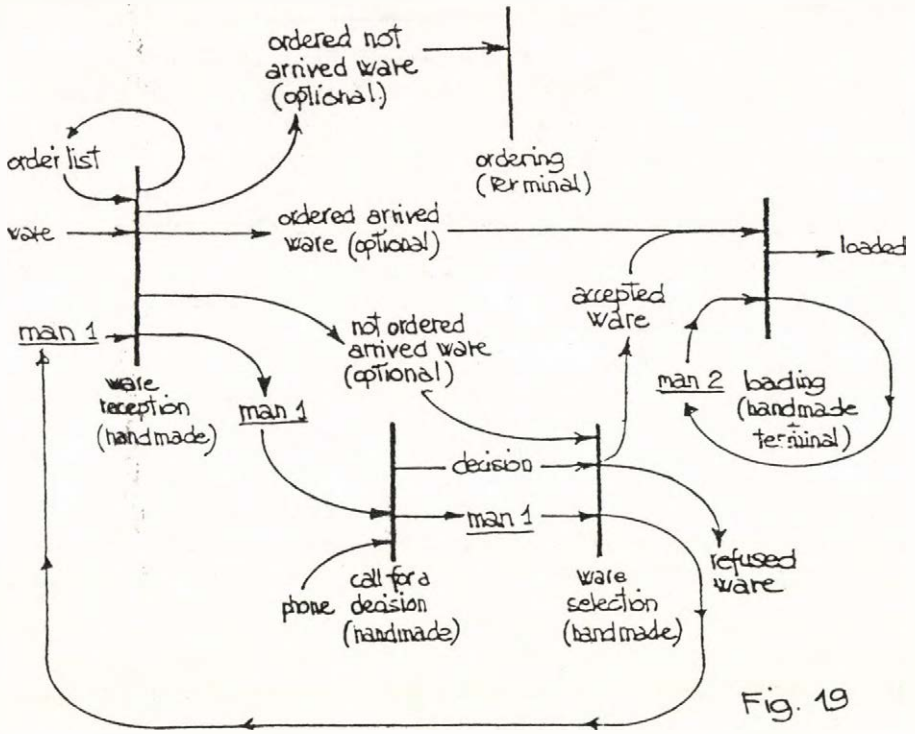


Fig. 19

which splits the activity into two steps performed by the resources man 1 and man 2, with the following table:

	ACTIVITY	MEAN HUMAN TIME SPENT	% OF THE OCCURIENCE	WEIGHTED TIME
Man 1	Reception	45'	100	45'
	Call	45' + 5'fix	15	≃ 12'
	Selection	30'	15	≃ 5'
TOT.1				≃ 1h
Man 2	Loading	2h	100	2h
TOT.2				2h

In this case, 7 arrival in a day can be carried on with three men, the first operating as resource man 1 and the remainder operating as resource man 2 and, furthermore, the latest arrival can be delayed; the net shows clearly the possible concurrencies among the various activities, allowing the choice of the best solution and a quantitative evaluation of the operational advantages in the use of the computers.

3.6 Petri nets, functional specification and design

A good functional specification should be completely independent by the design of the system, allowing then the tailoring of the architectural choices on the specific hardware or software; nevertheless, a sketch of the proposed architecture must be introduced in the document for planning purposes.

The shown method, involving "complex" but not "difficult" problems can be considered simultaneonsly the definition of the functional architecture of the system and the definition of the physical structure of the implementation, as can be derived by the operations which reach the description of the programs.

This fact is emphasized in the applications which are developed as a set of "transactional routines" (generally reentrant) managed by a specific added system monitor, whose purposes are the best suspend/restart operation for each terminal, the resources sharing, etc.

The method depicted above can be used until the refinements reach the level in which all the video "masks" have been described, being each transition of the net of that level a "transactional routine".

Let we se the example in Fig. 20

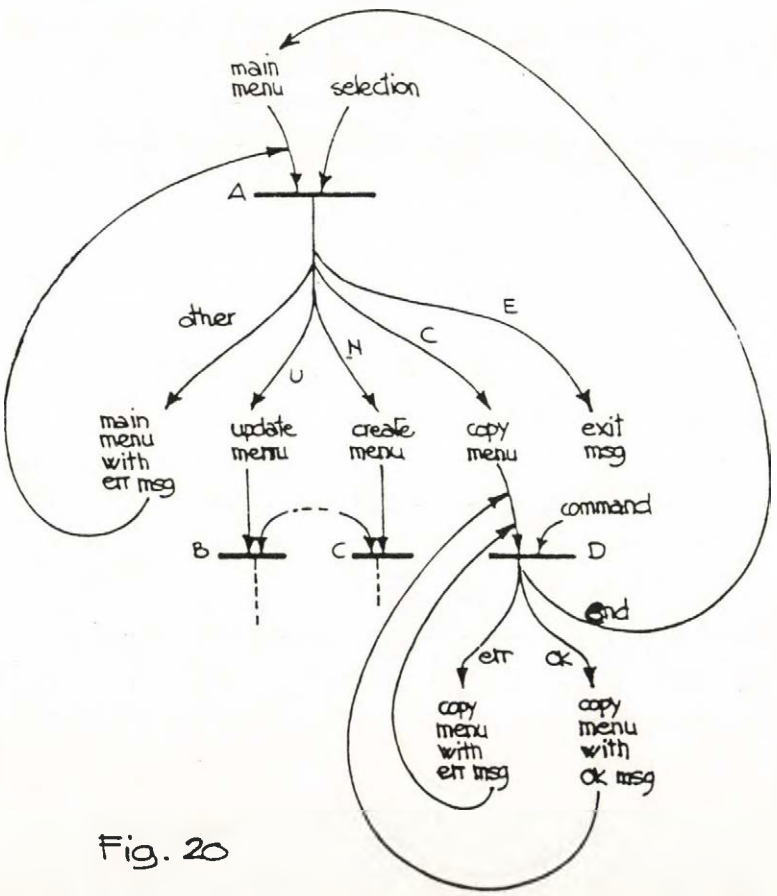


Fig. 20

The example, partial, refers to a (simplified) package for office automation; initially a "main menu" is provided to the user, which select a specific command. An 'E' indicates the end of the activity; a correct command activates the selected operation; an erroneous selection provides a message, allowing a new selection.
When the option 'C' is selected, the "copy menu" takes place, allowing the introduction of the proper data: uncorrect data induce error message and the retry, correct data induce the operation and the return to the "copy menu", the "end" selection returns to the main menu.

The net shows clearly the structure of the dialogs and the architecture of the functions (instead of a model based on finite state automata, typical for this kind of applications, but not expressive for the operational aspects), but is also the physical architecture: the main characteristics of the "transactiona routines" can be sommarized in a table such as the following:

TRANSACTIONAL ROUTINE	Starting Mask	events	next routine	output	next mask
A	main menu	E	-	exit msg	-
		C	D	-	copy menu
		N	C	-	create menu
		U	B	-	update menu
		other	A	err.msg	main menu
.....	....	....	....	....	....
D	copy menu	end	A	-	main menu
		correct	D	OK msg	copy menu
		errors	D	err msg	copy menu

4. Acknowledgements

The method is origined from the contribution of many people, amoug which particular thanks are due to G. Castelli, A.Cazziol, G. Degli Antoni, G. Haus, R.Polillo, B.Zonta.
It has been applied in many medium sized projects by M.Maiocchi and A. Cazziol of the Etnoteam S.p.A., by O. Sticchi of the TEMA S.p.A., by Bianchi of Parmalat and many others.

Requirements and functional specifications in the above experiences war related to systems devoted to medical analysis laboratories management, cash and carry invoicing procedures, book-keeping procedures, ware distribution companies management, budget procedures, management and billing for water and gas distribution in government owned companiers, and so on.

Evaluation application on the timing and the throughput of the system and on the resources allocation are mainly due to A. Cazziol, which turned the results in costs/benefits analysis.

5. References

1. J.L. Peterson - Petri nets - ACM Computing Surveys - vol.9, n.3, 1977

2. G. Castelli- M.Maiocchi - A methodology for the construction and the verification of functional specifications for Software procedures and programs - HIS Software Production Conference - Bloomington Minnesota - March 1979

3. A. Cicu, G. Degli Antoni, M. Maiocchi, R. Polillo, G.Torriani - An integrated multilevel documentation approach - Successful Software Management Techniques Conference - Bloomington (Minnesota), March 1978

4. G. Castelli, M. Maiocchi, G. Haus - Verso unametodologia per la costruzione di specifiche strutturate e corrette di procedure e programmi - Congresso AICA 1979 - Bari, ott. 1979

5. G. Degli Antoni, M. Maiocchi, R. Polillo, B. Zonta - How, What, Why - 4th HIS International Software Conference - Bloomington 1980

6. M. Maiocchi - Esperienze nell'uso di Reti di Petri perla definizione di specifiche funzionali - Giornata di lavoro su problemi di definizione di requirements, analisi e disegno di sistemi software - Obiettivo METOD - Progetto Finalizzato per l'Informatica - CNR - 1980

7. S. Cappelli - L'uso di Reti di Petri nella specifica funzionale di programmi: un esempio reale - Obiettivo - METOD - CNR 1980.

NONPROCEDURAL SPECIFICATIONS OF HARDWARE

Hans Eveking

Institut für Datentechnik
Technische Hochschule Darmstadt
D-6100 Darmstadt, Fed. Rep. of Germany

CONTENTS

CHAPTER 1

HARDWARE SPECIFICATION TECHNIQUES

1.1 Introduction

In the hardware design process, many levels coexist, for example, the circuit, timing, gate, register-transfer and microprogramming level. A group of items at one level is reperceived as a single "chunk" at a higher level [1]. A network of gates, for instance, is viewed as a single flipflop at the register-transfer level.

Assume that a hardware system is represented by a series of descriptions d(1), ..., d(n) where each description corresponds to some level (see Fig. 1). The description d(n) at the top-level is viewed as the specification given to the user of the hardware system. The specification of a microprogram instruction-set given to a programmer may serve as an example. The description d(1) at the bottom-level represents the implementation of the piece of hardware by means of the most basic resources, for example, by means of a network of gates. Moreover, each pair of descriptions d(i)/d(i-1), $1 < i \leq n$, is viewed as a specification/implementation pair of descriptions where d(i-1) is considered to be the implementation of the specification d(i).

Once we have decided to represent a hardware system by a series of descriptions d(1), ..., d(n) at distinct levels, we are faced with a number of problems. How can we show, for example, that d(1) finally meets its specification d(n) ? And how can we make sure that the descriptions d(1), ..., d(n) actually represent the same piece of hardware ?

In many applications, the descriptions d(i) are written in distinct and unrelated languages; some of the descriptions d(i) - in particular at the higher levels - may even be stated infomally (formal descriptions prevail at the bottom levels, e.g. the drawings, where preciseness is necessary for the manufacturing process). The verification of the specification d(n) is then based on the simulation of a number of test cases at the lower levels.

In this paper, we will discuss a different approach where it is assumed that all descriptions d(i) are written in some formal language L(i). If these languages have a common kernel then we are in principle able to prove by formal means that d(1) meets d(n). In the next section, we will briefly survey the requirements of such an approach.

1.2 The Requirements Of A Hardware Specification Technique

A description d(i) can be studied in two different ways (see Fig. 1):

1. We examine what the description d(i) does; the "what" is defined by the semantics of the language L(i) in which d(i) is written. For example, a reasoning process about d(i) can be based on the axiomatization of L(i). Note that the precise definition of the semantics of each language L(i) is crucial because a description d(i), in particular the description d(n) provided to the user, can be studied independently of the implementational details only if the semantics of L(i) is known.

2. We examine how d(i) is implemented by the description d(i-1) at the level below, for example, if we discuss how a flipflop is implemented by means of a network of gates. Note that the semantics of L(i) and L(i-1) must be given and must have a common "kernel": otherwise we are not able to relate descriptions in L(i) and L(i-1).

As a first requirement, we have thus to define the semantics of all languages L(i). The CONLAN approach [2, 3] provides a means for the precise definition of the semantics of a family of computer hardware description languages (CHDL's) which are intended to cover a variety of levels. All members of the family are derived from the single root language BASE CONLAN. In Chapter 2, the axiomatization of a small CHDL derived from BASE CONLAN will be discussed. We will confine our discussion to nonprocedural descriptions that prevail the gate, register-transfer and microprogramming level. A number of concepts specific to the nonprocedural nature of these descriptions will be presented.

As a second requirement, we have to show how descriptions at distinct levels are related on the basis of an axiomatization. The axioms associated with a description d(i) determine the correct statements that we can make about d(i), i.e. those statements which can be derived within the axioms. Although many details may be lost at some higher level j, a description d(j) at the higher level is still correct w.r.t. d(i) if the axioms associated with d(j) are correct statements about d(i). This approach will be discussed in Chapter 3.

CHAPTER 2

THE AXIOMATIZATION OF NONPROCEDURAL DESCRIPTIONS

2.1 Some Properties Of Nonprocedural CHDL's

Reasoning about a program in a common programming language - a methodology which goes back to the work of Hoare [4] and Floyd [5] and which is now represented by two books of Dijkstra [6] and Gries [7] - is quite different from reasoning about a description written in a nonprocedural CHDL. We will briefly review some of the basic principles of nonprocedural hardware descriptions:

1. Nonprocedural descriptions do not have a termination property, i.e. they describe the signal flow in a piece of hardware and not a computation (in the CONLAN frame-of-reference, we therefore talk about "descriptions" rather than "programs"). Consequently, the notions of partial and total correctness, pre- and postconditions do not apply to nonprocedural descriptions.

2. The statements of nonprocedural descriptions are executed in parallel. A locus of control is not implied; for example, the values of x and y are interchanged by the following two statements at each step of execution:

```
x<- y;
y<- x;
```

3. Nonprocedural CHDL's can be divided into step-time and real-time languages. Real-time CHDL's provide a delay-operator by which the delay of combinational circuits and the delay of lines can be modelled. Step-time CHDL's, e.g. the classical register-transfer languages like [8], are intended to model the behavior of finite automata. Step-time CHDL's do not have a delay operator; rather, an implied unit delay is provided to reflect the state before and after each state transition.

2.2 Time Functions

Our key concept for the axiomatization of nonprocedural descriptions will be the notion of a time function (see the concept of time functions in mathematical systems theory, e.g. [9]). The time will be represented by the set T,

$$T = \{U\} \cup N$$

where N is the set of nonnegative integers. Thus, T = {U, 0, 1, ...}. A time function is a function with domain T. The range of a time function f given by some set V defines the type of f, for example, a time predicate has range V = {FALSE, TRUE}, a ternary time function has range {U, 0, 1}, and a time operation has a subset of T as range. In the following Figure, a time predicate p, a ternary time function f and a time operation a are shown:

t	U	0	1	2	3	4	...
p(t)	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	...
f(t)	U	1	1	0	0	1	...
a(t)	U	U	1	1	1	4	...

Adopting the lambda-calculus, we define a time function f by

$$f = \text{lambda(t) (...)}$$

where the dots indicate the body of the function. The application of the time function f to some element t1 of T is denoted by f(t1). If f is a time function and a is a time operation then f(a(t1)) denotes the application of the concatenation of f and a to t1. In many cases, we will define time functions by means of expressions. The lambda expression

$$\text{lambda(t) (f(t) = g(t))},$$

for example, denotes a time predicate which is TRUE at interval t if the time functions f and g are equal at interval t. The application of this time function to interval 4 of time, for example, is denoted by:

$$\text{(lambda(t) (f(t) = g(t))) (4).}$$

As a basis for conditional definitions, we will employ the IF-THEN-ELSE-ENDIF construct. The boolean and-operation, for instance, is defined by:

```
and = lambda(p, q) (IF p THEN q ELSE FALSE ENDIF).
```

Rather than

```
IF p1 THEN q1
     ELSE IF p2 THEN q2 ...
                ELSE IF pn THEN qn ENDIF ... ENDIF
```

we will write:

```
IF p1 THEN q1 ELIF p2 THEN q2 ... ELIF pn THEN qn END.
```

2.2.1 Time Predicates

To denote time predicates, the logical operations & (and), | (or), ~ (not) and => (implication) will be used. We prefer standard infix notation and will write p & q rather than and(p, q), for instance. The following example shows the two time predicates p and q and the time predicate lambda(t) (p(t) & q(t)):

t	U	0	1	2	3	4	...
p(t)	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	...
q(t)	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	...
p(t)&q(t)	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	...

Definitions:

1. A time predicate p is a theorem iff p is TRUE for all elements t of T:

   (A t: p(t)).

   The time predicate lambda(t)(~(p(t)&q(t)) = ~p(t) | ~q(t)), for example, is a theorem due to De Morgan's law.

2. Two time functions f and g are equal iff the time predicate lambda(t) (f(t) = g(t)) is a theorem.

2.2.2 Time Operations

The usual arithmetic operations are extended to cope with the undefined element U of the time set T. The subtraction, for example, is defined as follows:

```
time_sub = lambda(t1, t2) (IF t1=U THEN U
                           ELIF t2=U THEN U
                           ELIF integer_less(t1, t2) THEN U
                           ELSE integer_subtract(t1, t2) END).
```

In this definition, integer_less and integer_subtract are the common arithmetic operations with integers. We prefer the infix notation "t1-t2" rather than "time_sub(t1, t2)". Similarly, the addition in the time set T is defined.

2.3 The Semantics Of Some Basic Language Constructs

In this section, the semantics of a number of constructs of the nonprocedural CHDL SMAX (small and axiomatized) will be defined using the concept of time functions introduced in the former section. SMAX is a very small CHDL derived from BASE CONLAN.

2.3.1 The Semantics Of Expressions

In the CONLAN approach, a piece of hardware is described by means of a set of carriers. Carriers are virtual points of observation in a piece of hardware. Representing the history of values that can be observed at a carrier, we associate a time function x with each carrier x. SMAX provides carriers x of type ternary only, i.e. where

```
(A t: (x(t)=U) | (x(t)=0) | (x(t)=1)).
```

The symbols "&&", "||" and "~~" denote the and-, or- and not-operations in ternary logic, respectively. If x and y are carriers of type ternary then we associate with an expression "x && y", for example, the time function

```
lambda(t) (tand(x(t), y(t)))
```

where the tand function is defined in the following usual way:

```
tand = lambda(x, y) (IF (x=1) & (y=1) THEN 1
                     ELIF (x=0) | (y=0) THEN 0
                     ELSE U END)
```

Similarly, the ternary or- and not-operation tor and tnot, respectively, are defined.

Expressions are delayed by the "%" delay operator. Let tfe be the time function associated with an expression e. Then "e%n" where n is a nonnegative integer denotes the time function

```
lambda(t) (tfe(t-n)).
```

If e is given by "x && y", for instance, then tfe becomes lambda(t) (tand(x(t), y(t))). Hence, "(x && y)%n" denotes the time function

```
lambda(t) ((lambda(t) (tand(x(t), y(t)))) (t-n))
```

which is equal to the time function

```
lambda(t) (tand(x(t-n), y(t-n))).
```

SMAX descriptions provide an ASSERTIONS part [10] which consists of a number of predicates separated by ",". We are asserting that these predicates should be TRUE at each interval of time. The ASSERTIONS construct is a means to specify the assumptions that one part of a system makes about the other parts (see the ASSERTIONS construct of SPECIAL [11]). An example is given by the timing conditions of integrated circuits, for instance, the set-up time requirement on the data-input of a flipflop. To specify ASSERTIONS, SMAX provides the operations "&", "|", "~" and "=>" corresponding to the logical operations introduced in section 2.2.1. With a SMAX ASSERTION "p=>q", for example, we associate the time predicate

```
lambda(t) (p(t) => q(t)).
```

The SMAX predicate "sstable" test for stability. sstable(x, td) is TRUE if x was stable in the last td time units. sstable(x, td) denotes the time predicate

lambda(t) (stable(x, td, t))

where the stable-function is defined as follows:

stable = lambda(x, td, t)(A i: 0 < i ≤ td: x(t-i) = x(t)).

Assume an ASSERTIONS part

ASSERT p1, ..., pn ENDASSERT

and let tp1, ..., tpn be the time predicates associated with p1, ..., pn, respectively. Then the ASSERTIONS part denotes the time predicate:

lambda(t) (tp1(t) & ... & tpn(t)).

SMAX permits expressions to denote time operations, too. The SMAX function "time" denotes the identity time operation

lambda(t) (t).

For example, we may write an ASSERTION "(100 < time) => ~(x = U)" which requires x not to be undefined after 100 time units. The SMAX function "sdelta" is defined as follows: sdelta(p) denotes the time operation

lambda(t) (delta(p, t))

where the delta-function is recursively defined as follows:

```
delta = lambda(p, t) (IF t=U THEN U
                      ELIF p(t) THEN t ELSE delta(p, t-1) END).
```

delta returns the last time interval when the time predicate p was TRUE:

t	U	0	1	2	3	4	...
p(t)	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	...
delta(p,t)	U	U	1	1	1	4	...

Note that the domain of delta(p, t) defines the subset of T which consists of all elements t where either t=U or p(t)=TRUE.

2.3.2 The Semantics Of Conditional Assignments

While expressions denote time functions, conditional assignments and connections denote theorems. To model the properties of storage elements, e.g. flipflops, SMAX provides carriers of type ternary variable which have a retention property. With a single conditional assignment to a carrier x of type tvar:

```
DECLARE x: tvar ENDDECLARE
IF a THEN x:= y ENDIF;
... "/other statements that do not affect x/"
```

where a and y may be any expression, the following axiom is associated:

```
(A t: x(t) = IF t=U THEN U
             ELIF a(t)=1 THEN y(t)
             ELIF a(t)=0 THEN x(t-1)
             ELSE U END).
```

In this axiom, a, y and x stand for the time functions associated with a, y and x, respectively. In plain english, this theorem tells us that the value of x at interval t is U if t=U, i.e. in the initial state. If the condition a equals 1 at interval t, then x(t) is equal to y(t); if a(t) is 0 then the old value x(t-1) is retained in x; if a(t) is U then x(t) equals U, too. The following Figure gives an example:

t	U	0	1	2	3	4	...
a(t)	U	0	1	1	0	1	...
y(t)	U	U	1	0	1	1	...
x(t)	U	U	1	0	0	1	...

A series of n conditional assignments:

```
DECLARE x: tvar ENDDECLARE
IF a1 THEN x:= y1 ENDIF;
...;
IF an THEN x:= yn ENDIF;
... "/other statements that do not affect x/"
```

has the semantics:

```
(A t: x(t) = IF t=U THEN U
             ELIF (a1(t)=1)&(A i: 1 < i ≤ n: ai(t)=0) THEN y1(t)
             ...
             ELIF (an(t)=1)&(A i: 1 ≤ i < n: ai(t)=0) THEN yn(t)
             ELIF (A i: 1 ≤ i ≤ n: ai(t)=0) THEN x(t-1)
             ELSE U ENDIF).
```

Thus, the old value x(t-1) is retained if all conditions a1, ..., an are 0 at interval t. If a condition ai equals 1 and all other conditions are 0 then x(t) is equal to the corresponding yi(t). In all other cases, in particular if a collision occurs (two or more conditions are 1), x(t) becomes U. Examples of application will be given in section 2.3.5.

2.3.3 The Semantics Of Conditional Connections

Carriers of type ternary terminal are used to model combinational networks without retention property. A single conditional connection to a carrier z of type ttml:

```
DECLARE z: ttml ENDDECLARE
IF a THEN z.= y ENDIF;
... "/other statements that do not affect z/"
```

has the semantics:

```
(A t: z(t) = IF t=U THEN U
             ELIF a(t)=1 THEN y(t)
             ELSE U END).
```

Thus, a ternary terminal is unequal U only if the condition a equals 1:

t	U	0	1	2	3	4	...
a(t)	U	0	1	1	0	1	...
y(t)	U	U	1	0	1	1	...
z(t)	U	U	1	0	U	1	...

A series of n conditional connections:

```
DECLARE z: ttml ENDDECLARE
IF a1 THEN z.= y1 ENDIF;
...;
IF an THEN z.= yn ENDIF;
... "/other statements that do not affect z/"
```

has the semantics:

```
(A t: z(t) = IF t=U THEN U
             ELIF (a1(t)=1)&(A i: 1 < i ≤ n: ai(t)=0) THEN y1(t)
             ...
             ELIF (an(t)=1)&(A i: 1 ≤ i < n: ai(t)=0) THEN yn(t)
             ELSE U ENDIF).
```

Examples of application are shown in section 2.3.5.

2.3.4 Nested IF-THEN-ELSE-ENDIF Statements

Note that we did not separate the semantics of the IF-THEN-ENDIF construct from the semantics of the assignment and connection operations. Rather, we consider the conditional invocation of an operation as a unity. This is due to the fact that in many CHDL's the conditions and operations are much closer related than by the usual concept where a boolean condition determines if an operation is executed or not.

The semantics of nested IF statements, operations in the ELSE-part, etc. are explained as follows: A nested conditional assignment, for example:

```
IF a THEN IF b THEN x:= y END
```

has the semantics of:

```
IF a&&b THEN x:= y END.
```

A conditional assignment found in the ELSE-clause of an IF-statement:

```
IF a THEN ... ELSE x:= y ENDIF
```

has the semantics of:

```
IF ~~a THEN x:= y ENDIF.
```

An unconditional assignment is a special case of a conditional assignment where the condition a is 1 for ~(t=U). Similar rules apply to conditional and unconditional connections. The THEN- and ELSE-part may include more than one assignment or connection operation separated

by ",". For example, we can simply write

```
IF a THEN x:= y, z.= k ENDIF
```

which is short for

```
IF a THEN x:= y ENDIF;
IF a THEN z.= k ENDIF.
```

2.3.5 Description Templates And Instances

SMAX incorporates the description template definition and instanciation features of BASE CONLAN (see [2] for a detailed introduction). Description templates define types of hardware modules. The interface of a description is given by IN, OUT and INOUT parameters defining the inputs, outputs and bidirectional connections. Attributes of a description, e.g. delay attributes are specified in the ATTribute section. Local carriers are declared between keywords DECLARE ... ENDDECLARE. The statements in the body of a description are separated by ";". Finally, the ASSERTIONS are specified.

In Figure 2, two example description templates are given. The first description shows a simple NAND-gate with two inputs. The gate delay is modelled by an unconditional connection of the output to the delayed inputs. The second description shows a rising-edge triggered d-flipflop. The storage property is represented by a conditional assignment to a local carrier ff of type tvar. The propagation delay is modelled by an unconditional connection of the output x to the internal carrier ff delayed by tp time units. The ASSERTION requires the data input y to be stable for a set-up time of tsu time units before the rising-edge of the signal a.

An instance inst of a description template templ is instanciated by means of the USE-statement:

```
USE inst(... "/actual IN/OUT/INOUT params./") :
             templ(... "/actual attributes/") ENDUSE
```

In Fig. 3, two instances of dff and nand2 (see Fig. 2), respectively, are used within a description ctrans.

2.3.6 The Axioms Associated With A Description

By means of the techniques introduced in the last sections, we associate with the set of statements of a description d a set of axioms. Any conclusion on the description d will be based on these axioms. In Fig. 4, the axioms associated with the descriptions dff and nand2 of Fig. 2 are given. In addition, the time predicate associated with the ASSERTION of dff is shown.

Assume an instance inst of a description template templ:

```
USE inst: templ ENDUSE
```

The axioms of the description instance inst are derived from the axioms associated with the description template templ as follows: Replace in the axioms associated with templ

1. all local carrier names by names qualified by the name of the instance, for example, replace the local carrier x by inst.x.

2. all formal IN and OUT parameters as well as attributes by actuals. If actuals are not specified in the USE statement then use qualified formals.

In Fig. 5, the axioms of the description ctrans of Fig. 3 are derived from the axioms of dff and nand2.

2.4 Language Constructs For Step-time Descriptions

In the last sections, we have introduced a number of language constructs for nonprocedural descriptions in order to model the behavior of a piece of hardware in real-time. The specification given to the user of such a piece of hardware, for example, the microprogram instruction-set obeys in general a different concept of time. In the example, the behavior of the machine is defined by a set of microprogram instructions each relating the state of the machine before and after the execution of one microinstruction; the microprogrammer is never allowed to refer to the state three microinstructions before, e.g. by means of a delay operator !

Descriptions with a simple before/after concept of time are called descriptions in step-time. From an implementation point of view, descriptions in step-time are intended to model a system at selected intervals of the real-time only and introduce thus a temporal abstraction (see section 3.3). The real-time intervals are selected by means of a reference signal, typically by means of a clock signal.

To provide a language for step-time descriptions, we will adopt the language constructs of sections 2.3 with the following exceptions:

1. The delay operator "%" and functions like sstable and sdelta that do not make sense in a step-time environment are removed.

2. Carriers of type tvar are not adequate to reflect the before/after relation. The type tvar is replaced by a new carrier type tudv (ternary unit delay variable) where a unit delay of the transfer condition as well as of the source is assumed. Rather than

```
DECLARE x: tvar ENDDECLARE
IF a%1 THEN x:= y%1 ENDIF, ...
```

we will simply write:

```
DECLARE x: tudv ENDDECLARE
IF a THEN x<- y ENDIF; ...
```

The following axiom is associated with a single conditional transfer:

```
(A t: x(t) = IF t=U | t=0 THEN U
             ELIF a(t-1)=1 THEN y(t-1)
             ELIF a(t-1)=0 THEN x(t-1)
             ELSE U END).
```

The following Figure gives an example of the behavior of the conditional transfer:

t	U	0	1	2	3	4	...
a(t)	U	0	1	1	0	1	...
y(t)	U	U	1	0	1	1	...
x(t)	U	U	U	1	0	0	...

A series of n conditional transfers:

```
DECLARE x: tudv ENDDECLARE
IF a1 THEN x<- y1 ENDIF;
...;
IF an THEN x<- yn ENDIF;
... "/other statements that do not affect x/"
```

has the semantics:

```
(A t: x(t) =
      IF t=U | t=0 THEN U
      ELIF (a1(t-1)=1)&(A i: 1 < i ≤ n: ai(t-1)=0)
           THEN y1(t-1)
      ...
      ELIF (an(t-1)=1)&(A i: 1 ≤ i < n: ai(t-1)=0)
           THEN yn(t-1)
      ELIF (A i: 1 ≤ i ≤ n: ai(t-1)=0) THEN x(t-1)
      ELSE U ENDIF).
```

3. For the specification of microprogram instruction sets, the ACTIVITY declaration and invocation feature of BASE CONLAN will be adopted in a very restricted way. In SMAX, activities are viewed as parametrized macros's. Local carriers must not be declared in the body of an activity. Carriers of the enclosing description segment may be imported via the IMPORT statement. The invocation of an activity simply means the textual substitution of the activity's body where formal parameters (if any) are replaced by actual ones. For an example, see Fig. 10 which will further be discussed in section 3.3

CHAPTER 3

REASONING ABOUT NONPROCEDURAL DESCRIPTIONS

3.1 Inference Rules

In a reasoning process about a hardware description, we will conclude that a time predicate is a theorem from the axioms associated with the hardware description. A number of inference rules guide the reasoning process. We will write inference rules in the form:

$$\frac{a1, \ldots, an}{c}$$

which has the following meaning: If the antecedents a1, ..., an are theorems then so is the consequent c. Examples of application of the following rules will be given in the next section.

1. Let f and g be time functions and let a and b be time operations. Moreover, let P(string) denote some time predicate P in which string is found and let P(x/string) denote the time predicate P where some free occurrences of string are replaced by x. Then the following rule applies (substitution of equals for equals):

$$\frac{(\mathbf{A}\ t:\ f(a(t)) = g(b(t)))}{(\mathbf{A}\ t:\ P(f(a(t))) = P(g(b(t))/f(a(t))))}$$

2. Let P(string) denote again a time predicate P in which string is found. Let P(string//x) denote the time predicate P where <u>all</u> free occurrences of string are replaced by x. Let a be a time operation. Then the following rule holds:

$$\frac{(\mathbf{A}\ t:\ P(t))}{(\mathbf{A}\ t:\ P(a(t)//t))}$$

The second rule is justified because (a) if P(t) is a theorem then it is TRUE for all elements of T and (b) the time operation a has a range which is a subset of T.

3. The third rule deals with induction over the time set T. Let P(t) denote a time predicate P with the time variable t and let P(x//t) denote P where all occurrences of t are replaced by x. Then

$$\frac{P(U//t),\ P(0//t),\ P(t) \Rightarrow P(t+1//t)}{(\mathbf{A}\ t:\ P(t))}$$

If P is TRUE for intervals U and 0 and if from P(t) follows that P is TRUE for interval t+1, then we conclude that P is TRUE for all elements of T.

3.2 Correct And Equivalent Descriptions

Definitions:

1. A time predicate p is a correct statement w.r.t. a description d if the theoremhood of p can be concluded from the axioms associated with d.

2. A description d(j) is correct w.r.t. a description d(i) if the axioms associated with d(j) are correct statements w.r.t. d(i).

3. Two descriptions d(j) and d(i) are equivalent if d(j) is correct w.r.t. d(i) and if d(i) is correct w.r.t. d(j).

4. A description d(j) is a correct description of d(i) w.r.t. a representational function phi if the axioms of d(j) mapped by phi are correct statements w.r.t. d(i).

We will first give some examples of correct statements w.r.t. a description. The statement x(U)=U, for instance, is correct w.r.t. the description delayel of Fig. 6. which follows immediately from the axiom associated with delayel. Moreover, the statement

$$(\mathbf{A}\ t:\ x(t) = y(t-n))$$

is correct w.r.t. delayel: if t=U then t-n = U according to the definition of subtraction given in section 2.2.2 and thus y(t-n) = y(U) = U because y is assumed to be a carrier of type ttml.

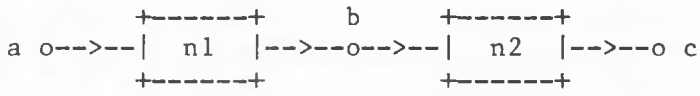
A further example is given by the statement

$$(\mathbf{A}\ t:\ c(t) = a(t-n1-n2))$$

which is correct w.r.t. the description twodelayel of Fig. 7. Proof: On account of the former example,

$$(\mathbf{A}\ t:\ b(t) = a(t-n1))$$
$$(\mathbf{A}\ t:\ c(t) = b(t-n2))$$

holds for twodelayel:



Due to Rule 2 of the former section, t can be replaced by t-n2 in the first theorem yielding

$$(A\ t:\ b(t-n2) = a(t-n1-n2)).$$

Following Rule 1, we replace b(t-n2) by a(t-n1-n2) in the second theorem which proves our statement.

An example of a correct description is given by the description doubledelay (Fig. 8) which is correct w.r.t. the description twodelayel of Fig. 7. However, doubledelay and twodelayel are not equivalent because the axiom associated with the carrier b of twodelayel can not be derived from the axiom associated with doubledelay.

The description twodelayell of Fig. 9 corresponds to the description twodelayel of Fig. 7 with one exception: the local carrier b is replaced by the local carrier x. Defining a representational function phi by

$$phi(x(t)) = b(t)$$

we can show that twodelayell is a correct description of twodelayel w.r.t. phi. Representaional functions are used if, in the specification, a carrier or a concept of time is introduced which is not found in the implementation.

3.3 Abstract Descriptions

We consider an abstract description d(j) to be a correct, less-detailed description w.r.t. a description d(i). In the hardware design process, at least three different kinds of abstraction are found by which a description d(j) may be less-detailed than d(i):

1. Spatial abstraction: some carriers are found in d(i) but not in d(j). The descriptions doubledelay and twodelayel of Figs. 8 and 7 give an example of spatial abstraction: the carrier b of the description twodelayel is not found in doubledelay.

2. Abstraction by ASSERTIONS: some conditions which may occur in d(i) are excluded by ASSERTIONS of d(j). An example of abstraction by ASSERTIONS is given by the description dff of Fig. 2 which is intended to model a "real" flipflop. The "real" flipflop behaves anyway even if the set-up time requirement in the ASSERTIONS part of dff is not satisfied. For example, the flipflop may assume a meta-stable state which is however very difficult to model. We therefore exclude this undesired situation by means of an ASSERTION; the simplified behavior specified in the body of dff holds only if the ASSERTION is satisfied.

3. Temporal abstraction: The behavior of d(i) is only partially defined by d(j) in the time-dimension. This kind of abstraction is used if descriptions in real-time are

represented in step-time.

We will finally apply these concepts of abstraction to a simple example where we assume a microprogrammable machine to be described at three distinct levels. The microprgram instruction set is specified in step-time (description d3 of Fig. 10) relating the state of the machine before and after the execution of one microprogram instruction. The machine is described at the register-transfer level in step-time providing a functional description of the main registers, busses and combinational networks (description d2 of Fig. 10). Finally, the network of integrated circuits where the behavior of each IC is represented in real-time is shown (see Fig. 11).

The concept of spatial abstraction is applied in the descriptions d3 and d2 of Fig. 10. A number of carriers of d2, e.g. the ROM, the decoders, etc. are not found in d3. The intention of d3 is to state the effect of a microinstruction (and a microprogram) on those carriers only which are accessible to the microprogrammer. Implementational details which are invisible for the microprogrammer are not represented.

Fig. 11 shows how one of the conditional transfers of d2 is implemented by means of an IC network given by the description ctrans of Fig. 3. In this example, all concepts of abstraction are mixed:

1. A number of carriers of the IC network are not represented by d2, for example, the clock signal (spatial abstraction).

2. A number of conditions are excluded by ASSERTIONS.

3. The step-time description d2 (as well as d3) is intended to model the behavior of the IC network only at those intervals of time where the clock signal has a falling edge (temporal abstraction). Step-time intervals are related to real-time intervals by means of a representational function (Fig. 12). The delta-function used in the definition of phi selects the intervals where the clock signal has a falling edge. The description d2 is a correct description of d1 w.r.t. phi if the additional ASSERTIONS of d1 are satisfied. The proof involves the application of the induction theorem of section 3.1 (see [10]).

SUMMARY

In this paper, the axiomatization of nonprocedural hardware descriptions was discussed. The axiomatization of descriptions at distinct levels provides a common semantical kernel by which descriptions at distinct levels are related. The concepts of correct and abstract descriptions were introduced and a number of examples of application were given.

REFERENCES

[1] Hofstadter, D.R.: Godel, Escher, Bach: An Eternal Golden Braid, Vintage Books, New York 1980.

[2] Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F. and Skelly, P.: CONLAN Report, Springer, Berlin Heidelberg New York Tokyo, 1983.

[3] Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F. and Skelly, P.: CONLAN - A Formal Construction Method for Hardware Description Languages: Basic Principles, Language Derivation and Language Application (3 papers), Proc. NCC, Vol. 49, Anaheim 1980.

[4] Hoare, C.A.: An Axiomatic Basis for Computer Programming, CACM, Vol. 12, 1969, pp. 576-581.

[5] Floyd, R.: Assigning meaning to programs, Mathematical Aspects of Computer Science, Vol. 19, 1967, pp. 19-32.

[6] Dijkstra, E.W.: A Discipline of Programming, Prentice Hall, 1976.

[7] Gries, D.: The Science of Programming, Springer, New York/Heidelberg/Berlin, 1981.

[8] Chu, Y.: An ALGOL-like computer design language, CACM Vol. 8, 10/1965, pp. 607-615.

[9] Windeknecht, Th.G.: General Dynamic Processes, Academic Press, New York/London, 1971.

[10] Eveking, H.: The Application of CONLAN ASSERTIONS to the Correct Description of Hardware, Proc. 5th Int. Symp. on CHDL's, pp. 37-50, Kaiserslautern, 1981.

[11] Roubine, O., Robinson, L.: SPECIAL Reference Manual, SRI Technical Report, 1976.

FIGURES

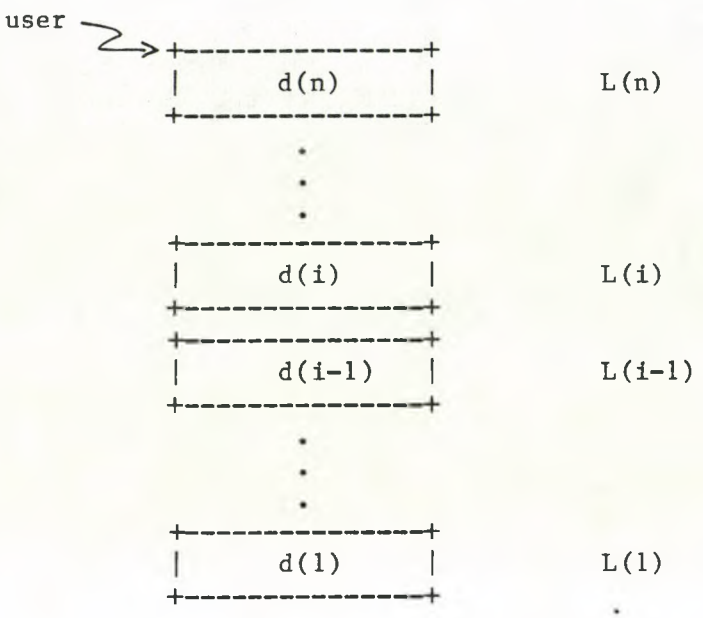


Fig. 1: A hardware system represented by a series of descriptions at different levels

```
DESCRIPTION nand2(ATT dnand: int)
                (IN y, z: ttml; OUT x: ttml)
                BODY x.= ~~(y && z)%dnand
ENDnand2


DESCRIPTION dff(ATT tp, tsu: int)
              (IN y, a: ttml; OUT x: ttml)
              BODY DECLARE ff: tvar ENDDECLARE
              IF a && ~~a%1 THEN ff:= y ENDIF;
              x.= ff%tp
              ASSERT
              ((a && ~~a%1)=1) => sstable(y, tsu)
              ENDASSERT
ENDdff
```

Fig. 2: The description templates dff and nand2

```
DESCRIPTION ctrans(ATT tp, tsu, tpnand: int)
                  (IN clock, y, cond: ttml; OUT x: ttml)
                  BODY DECLARE int: ttml ENDDECLARE
                  USE nand2inst(clock, cond, int): nand2(tpnand);
                      dffinst(y, int, x): dff(tp, tsu) ENDUSE
ENDctrans
```

Fig. 3: The description ctrans consisting of one instance of dff and one instance of nand2

```
(A t: x(t) = IF t=U THEN U
             ELSE tnot(tand(y(t-dnand), z(t-dnand))) ENDIF)
```

```
(A t: ff(t) = IF t=U THEN U
              ELIF tand(a(t), tnot(a(t-1)))=1 THEN y(t)
              ELIF tand(a(t), tnot(a(t-1)))=0 THEN ff(t-1)
              ELSE U ENDIF)
(A t: x(t) = IF t=U THEN U
             ELSE ff(t-tp) ENDIF)
lambda(t) ((tand(a(t), tnot(a(t-1)))=1) => stable(y, tsu, t))
```

Fig. 4: The axioms and the time predicate associated with the descriptions nand2 and dff of Fig. 2

```
(A t: int(t) =
      IF t=U THEN U
      ELSE tnot(tand(clock(t-dnand), cond(t-dnand))) ENDIF)
(A t: dffinst.ff(t) =
      IF t=U THEN U
      ELIF tand(int(t), tnot(int(t-1)))=1 THEN y(t)
      ELIF tand(int(t), tnot(int(t-1)))=0 THEN dffinst.ff(t-1)
      ELSE U ENDIF)
(A t: x(t) = IF t=U THEN U
             ELSE dffinst.ff(t-tp) ENDIF)
lambda(t) ((tand(int(t), tnot(int(t-1)))=1) => stable(y, tsu, t))
```

Fig. 5: The axioms and the time predicate associated with the description ctrans

```
DESCRIPTION delayel(ATT n: int)
                    (IN y: ttml; OUT x: ttml)
            BODY x.= y%n
ENDdelayel

(A t: x(t) = IF t=U THEN U ELSE y(t-n) END)
```

Fig. 6: The description template delayel and its axiom

```
DESCRIPTION twodelayel(ATT n1, n2: int)
                      (IN a: ttml; OUT c: ttml)
            BODY DECLARE b: ttml ENDDECLARE
            USE del1(a, b): delayel(n1);
                del2(b, c): delayel(n2) ENDUSE
ENDtwodelayel

(A t: b(t) = IF t=U THEN U ELSE a(t-n1) END)
(A t: c(t) = IF t=U THEN U ELSE b(t-n2) END)
```

Fig. 7: The description twodelayel and its axioms

```
DESCRIPTION doubledelay(ATT n1, n2: int)
                       (IN a: ttml; OUT c: ttml)
            BODY c.= a%(n1+n2)
ENDdoubledelay

(A t: c(t) = IF t=U THEN U ELSE a(t-n1-n2) ENDIF)
```

Fig. 8: The description doubledelay and its axioms; doubledelay is correct w.r.t. the description twodelayel of Fig. 7

```
DESCRIPTION twodelayel1(ATT n1, n2: int)
                       (IN a: ttml; OUT c: ttml)
            BODY DECLARE x: ttml ENDDECLARE
            USE del1(a, x): delayel(n1);
                del2(x, c): delayel(n2) ENDUSE
ENDtwodelayel1
```

Fig. 9: twodelayel1 is a correct description of twodelayel w.r.t. the representational function phi: phi(x(t))=b(t).

```
DESCRIPTION d3 ... BODY
     DECLARE a, b, ...: tudv; bus, madr, ...: ttml ENDDECLARE
     "/specification of the microprogram instruction set:/"
     ACTIVITY loada BODY
              IMPORT a, bus ENDIMPORT
              a<- bus
     ENDloada
     ACTIVITY loadb BODY
              IMPORT b, bus ENDIMPORT
              b<- bus
     ENDloadb
     ...
          "/specification of the microprogram,
            madr = microprogram address:/"
          IF madr=100 THEN loada ENDIF;
          IF madr=101 THEN loadb ENDIF;
          ...
ENDd3


DESCRIPTION d2 ... BODY
     DECLARE a, b, ...: tudv; bus, rom0, rom1,
             decode0, decode1, madr, ...: ttml ENDDECLARE
     "/functional description of decoder and registers:/"
     decode0.= ~~rom0 && ~~rom1;
     decode1.= ~~rom0 && rom1;
     IF decode0 THEN a<- bus ENDIF;
     IF decode1 THEN b<- bus ENDIF;
     ...
          "/description of the microprogram as the content
            of a ROM, madr = ROM address:/"
          IF madr=100 THEN rom0.= 0, rom1.= 0 ENDIF;
          IF madr=101 THEN rom0.= 0, rom1.= 1 ENDIF;
          ...
ENDd2
```

Fig. 10: The description of the microprogram and the microprogram instruction set (d3) and the functional description at the register-transfer level (d2)

```
DESCRIPTION d2 ... BODY
                   IF decode0 THEN a<- bus ENDIF;
                   ...
ENDd2


DESCRIPTION d1 ... BODY
            USE registera(clock, bus, decode0, a):
                          ctrans(tp, tsu, tpnand) ENDUSE;
            ...
            "/ASSERTIONS that guarantee the correct
              implementation:/"
            ASSERT (clock%1&&(~~a)&&a%1)=0,
            ((~~clock&&clock%1)=1) => (a=a%1)&
            (sdelta((~~clock&&clock%1)=1)%1 ≤ tp+tpnand),
            ((~~clock&&clock%1)=1)%tpnand => (bus=bus%tpnand),
            ((~~clock&&clock%1)=U) =>
            (sdelta((~~clock&&clock%1)=1) = U)
            ENDASSERT
ENDd1
```

Fig. 11: A conditional transfer at the register transfer level and its implementation by means of the description ctrans of Fig. 3

```
(A tau: a(tau) = IF (tau=U)|(tau=0) THEN U
                 ELIF decode0(tau-1)=1 THEN bus(tau-1)
                 ELIF decode0(tau-1)=0 THEN a(tau-1)
                 ELSE U END)


     phi(tau) = delta((~~clock&&clock%1)=1, t)
     phi(tau-1) = delta((~~clock&&clock%1)=1,
                        delta((~~clock&&clock%1)=1, t)-1)
```

Fig. 12: The semantics of the conditional transfer of Fig. 11 referring to step-time intervals tau and the representational function phi mapping step-time intervals tau in real-time intervals t

On the Specification and Manipulation of Forms.

by

I. Balbin,
P.C. Poole,
and
C.J. Stuart

Department of Computer Science,
University of Melbourne,
Parkville 3052,
Melbourne,
Australia.

ABSTRACT

Electronic replicas of printed forms provide a very convenient man-machine interface for capturing data. The keyboard operator fills in the form under the control of the computer which can reject invalid data. In many systems, the form is simply an I/O filter and, while convenient mechanisms are provided for describing the layout of the form, it is still necessary to write a program to capture and validate data and sequence the user through the form. In this paper, we describe a high level language for specifying forms and a transaction processing system for manipulating them. The specification is complete in the sense that it contains sufficient information for the system to determine the source of the data, what data is valid, how it is to be displayed and the sequence of fields to be visited which may depend on data previously input and the state of the database. We argue that this approach will significantly reduce the cost of producing software for transaction oriented data processing applications.

1. Introduction

Over the last few years, a considerable amount of research has taken place into office automation. There are essentially two approaches which have been explored in order to provide an automated office. The first [12,15] involves the classic top-down design methodology which identifies the general overall requirements of an office and then seeks to automate these. This method, while attractive in theory, has not yet led to many practical implementations. The other approach [27-29] has been to identify common office functions. As the requirements for each of these crystallise through the exposure of prototypes to the real world, one can continually gain valuable insights which will eventually lead to integration.

We have been following the second approach focusing our research on forms [16], which are electronic replicas of the printed business forms commonly used for collecting information in the modern office. Some examples of printed forms which are in common use are cheques, tax returns and bank withdrawal forms. After filling in a form, a check is made to ascertain, as far as possible, that the information is correct, after which the form is filed in a database. The act of filing the form may itself trigger other activities such as updating the database.

FSL [1,2] (Forms Specification Language) is a very high level language for specifying and solving a variety of business data processing problems by describing forms. The emphasis is not on how to solve a problem but on what needs to be done. FSL provides convenient and powerful constructs and a natural language syntax for describing forms which are amenable to the office worker who is not necessarily a computer programmer and who might normally solve such a problem manually. A compiler translates the forms specification into a data structure which is interpreted by TPS [22], a Transaction Processing System, designed to capture and manage data from the user(s) by way of a forms-oriented interface. It has been implemented in C running under the UNIX[1] operating system.

A printed form consists of text and indicators showing where the user is to provide information. Similarly, a form specification in FSL describes displayed text and the entities called **fields** which correspond to the places where information is collected and displayed. In a completed form, fields have a content. In a printed form, these are invariably provided by the user filling in the field; using TPS, the content may be supplied by the user or generated by the system and may be confined to be in a particular domain.

It has been noted [11] that offices tend to be divided into organisational units such as divisions and departments. Each division uses a number of documents or forms through which its data processing is accomplished. We define such a related group of forms as a menu. The process of implementing a system for a particular application can then be loosely described as follows:

(1) Identify the principal menus to be used by the organisation.

(2) Further subdivide each menu into a list of forms (or menus).

(3) using FSL, specify the description of the fields comprising each form.

---

[1]UNIX is a Trademark of Bell Laboratories.

(4) compile the forms into tables suitable for interpretation by TPS.

(5) invoke TPS, interactively filling in any form in the menu via a cursor addressable display terminal.

It can be seen that the hierarchical organisation of forms models the top down design process used in the construction of a total application. In fact, this aspect of forms has been further developed and advocated in [14].

In section 2, we discuss some of the relevant literature. Section 3 describes FSL with appropriate examples and in sections 4 and 5, we present some implementation considerations with respect to TPS. Appendix A illustrates an example form template.

2. Related Research

There are essentially two complementary aspects in the specification of a form. One is the specification of the form fields themselves and the other is the specification of the organisational flow of forms amongst several workstations resulting from predefined trigger conditions. The former, which was loosely termed "integrity constraints on forms", has been explored in [9,17,19,21], whilst the latter, which has been termed "form management" or "forms manipulation", has been discussed in [7,24-26]. We note that both aspects are vital to the overall achievement of effective office automation. FSL is essentially a language which concentrates on the form field definition aspect of forms research. Unlike earlier systems, e.g. SQL/PL1 [4], which coupled a database management system with a general purpose programming language, FSL directly incorporates these features, at the same time, relieving the user of the task of dealing with the particular nuances of the DBMS. A comprehensive discussion of the shortcomings of these systems can be found in [3].

It is clear that the definition of a form includes such entities as documents, pads, memos [18] or slips of paper [6]. Thus, we do not consider it desirable to differentiate between them. The specification language for the form should be general enough to include all of these. There is some debate [7], however, as to whether the form has been taken out of its natural usage and coerced (as in OBE [29] for example) to implement functions which, although contributing to integration, are not necessarily the desirable solution. It can be argued that the condition and repetition boxes of OBE are foreign to an end user whereas an english-like statement is more appropriate.

FSL has been designed for the office worker and we have consequently made it non-procedural to keep it as simple as possible. Likewise, BUSINESS [18] has also been designed with similar users in mind. However, it is doubtful that the nested procedure, which is part of its syntax, is the "specification of a solution in terms already familiar to the end user".

SBA [5] and QBE/OBE [27-29] are systems which serve as an interface to the IBM query language QBE. Whilst QBE is relatively powerful [20] and easy to learn [23], much of the data processing in an application is done in a well structured and predetermined fashion which is performed

repeatedly. The transactions often modify the database in a variety of ways depending on user input or the data that is accessed. At the same time, we must admit that there are occasions when queries are desirable and that the FSL-TPS system does not, as yet, have a satisfactory facility to handle these.

3. Forms specification

One of the objectives of the FSL-TPS project was to produce a system which was data-driven. By contrast, most programs are procedure driven, which means that they follow a fixed sequence of actions. A data-driven system responds to events which cause changes in some data structure and the actions taken depend not so much on a fixed sequence but on the nature of the event. In writing a program in FSL, one does not write a set of procedures but, rather, provides a description of a data structure and how it depends on user-provided or database-provided information. This structure corresponds to a completed form which we refer to as a form instance. TPS is used to construct a form instance, an operation which corresponds to filling in the form, as it responds to the events of the user entering data or to alterations of the database.

There are two primary components in the specification of a form. The first consists of the form template, an example of which is given in Appendix A. This is the static text associated with a form type serving to identify the nature of the form and its fields. The second component is the specification of the individual fields comprising the form.

3.1. Form Template.

3.1.1. Heading.

A heading is a string of characters appearing on a form having no particular connection with the individual fields. In Appendix A, for example, the heading "REGULAR INVOICE" serves to identify the form to the user but has no apparent importance with respect to any individual field.

3.1.2. Prompt.

A prompt is a string of characters appearing on a form which serves to identify a particular field. In Appendix A, for example, the prompt "account no." indicates to the user that the field represents the account number for the invoice. Since a prompt is necessarily associated with a field, it is included in the field specification. The position, field identifier (if supplied) and prompt are called the location attribute of the field. Currently, the position, which is simply the row and column number on the screen, must be supplied by the forms designer as absolute or relative values. However, in an interactive forms editor currently under development, these positions will be calculated by the system.

3.2. Field Specification.

Once the location attribute has been specified, it is then necessary to specify the set of possible contents of the field. This is

termed the value attribute. From this specification, the system can determine the size of the field and the source of the data, that is, whether the content of the field will be input from the keyboard or generated automatically by the system, either by evaluating an expression or via an update after some other form has been filed. The way in which the field is to be displayed may also be specified.

3.2.1. Accepting Statement.

The content of a field may come from a number of sources, the most usual being from the keyboard when the user is interactively completing the form after invoking TPS. In all instances, however, the forms designer requires the capability to specify the set of inputs which are valid and which may consequently be filed. In FSL, the accepting statement performs this task.

3.2.1.1. Standard Types.

FSL provides a number of high level constructs which can be used to specify the acceptance criterion. The field specification for the "account no." in Appendix A can be written in FSL using one of these as follows :-

```
numbered field is prompted by "account no." at (2,20)
        accepting integer(5);
```

indicating that valid input is an integer in the range 0 to 99999. The source of the input is, by implication, the keyboard. The key word numbered causes the prompt to be preceded by a generated integer which can be used to select the field for amendment. This feature is only available for fields where input may come from the keyboard. The actual account number is called the key field and its content is used to identify the record created through the form in the data base for subsequent retrieval or updating.

In some circumstances, the forms designer may wish to specify that input from the keyboard is optional, i.e. that if characters are input, then they will be subjected to validation criteria, otherwise the null input is acceptable. This may be specified in FSL by :-

```
accepting alphabetic(10) or null ;
```

3.2.1.2. User Defined Types.

Often using a standard type is far too general. It is therefore also possible to include specific strings in the accepting statement. Consider a field representing a part number for a particular product. This part number consists of a string representing the year the part was manufactured, the actual stock number of the product, and a string representing the warehouse where the part is located. This field can be specified in FSL by -

```
field is prompted by "part no." at (3,30)
        accepting year:82..99 & '/' & integer(5) & alphabetic(2);
```

where '&' is the concatenation operator. The input string consists of four separate sections known as subfields. It is possible to extract

the content of a subfield, as in the year component above, by prepending an identifier to the field and subsequently using it in an expression.

3.2.2. Displaying Statement.

There is an important distinction to be made between the content of a field in the record and the string displayed on the screen for a field. In most cases, the forms designer will want these to be identical. However, there are instances when a suitable mapping from input to output is desirable. FSL supports this mechanism permitting a number of display formats. The displaying statement also allows the user to display a different value on the screen for that field.

3.2.3. Source.

In the examples seen thus far, the source of the input to the field has been implicitly specified to be the keyboard. In many instances, however, the content of a field is determined solely by evaluating an expression. In such cases, the key word assigning is used. Expressions are made up of operands, operators and calls on system functions. Operands may be fields that are not local to the current form. Consider, for example, a field representing the sales tax on a product :-

```
field is prompted by "Sales Tax." at (4,10)
        accepting real(6)
        assigning       (unit_price - discount) / 9.37  if year >  82
                  else (unit_price - discount - depreciation) / 9.37 ;
```

The content of the field is determined by a conditional expression which takes into account the age of the product and its consequent depreciation. In this instance, the content of the field is displayed automatically requiring no input from the user.

In addition to having fields which can derive their contents from expressions, it is also desirable to have a facility which allows this assignment to be overridden. For example, a field representing the discount due to a customer may depend on the number of items purchased from the particular sale as well as the number of items purchased over a period of time. The company may, however, want to give a different discount in exceptional circumstances. In other words, the company wishes to use the value of the expression in the default situation only. This may be specified by using the key words defaulting to instead of assigning.

3.2.4. Updated Fields

In the discussion thus far, the content of a field has either come from the keyboard or by evaluating an expression or a combination of both methods. There is one other important means by which the content of a field is determined and that is via updates initiated by the filing of other forms. An example could be a field representing the total number of products bought by a customer. The field is updated every time an invoice is lodged. A suitable specification for this field would be :-

```
field is prompted by "overall sales" at (7,7)
        accepting integer(5) updated from Invoices.key by
         overall_sales + Invoices.key.number_sold;
```

The field (which in this case is not local to the "Invoices" form) is updated every time an invoice is lodged using that customer's account number.

3.3. Aggregate Fields.

A facility for aggregating fields is provided in the form of a table which may be thought of as an array of fields. An exit condition can be attached to a field in such an aggregate which enables the user to avoid having to complete all the fields in the table at execution time. An example of a table given in Appendix A consists of the product name, description, quantity etc. This can be specified in FSL by :-

```
table with 6 entries {
        field Product is at (+1,3)
                accepting alphabetic(7) exiting if Product is '';

         /* other table fields come here */
```

3.4. Entry Condition.

After the location and value attributes for each field have been given, the specification of the form is complete apart from information which determines whether or not fields are relevant. An irrelevant field is one which is not considered in the completed form; for example, in the invoice form given in Appendix A, tax information may not be considered if the customer is a reseller, as defined by field 2. The designer may introduce entry conditions, which may be nested, to determine if fields are to skipped over as irrelevant. For example, the following FSL specification indicates that field_1 is to be ignored unless the condition is true.

```
field_Ø ......
enter if <condition> {
        field_1 is ......
}
field_2 is ......
```

After the content of field_Ø has been determined, TPS tests the entry condition. If this evaluates to true, then TPS will guide the user to field_1, otherwise the cursor will be directed to field_2.

4. Interpreting forms

The program TPS is used to interpret a set of forms and menus, written in FSL. The user may select a form under menu control, and then complete it as TPS captures, validates and displays the relevant data. The completed form may then be filed, causing the execution of the relevant updates. Furthermore, TPS allows several users to be accessing

and altering the database simultaneously, and ensures that the database remains consistent throughout. The algorithms used by TPS reflect the data driven nature of FSL.

4.1. The compiled form

The compiled form module, which we will refer to hereafter simply as the form, is arranged as a set of field descriptions. The task of TPS is to give each field a valid content and then to file the set of field contents, which is the form instance, into the database.

In the compiled form, each field has four attributes. These are:

(1) the content attribute which describes how the content of a field is obtained.

(2) the validity attribute which is used to determine whether or not a particular content is valid.

(3) the display attribute which describes how a field is presented to the user.

(4) the successor attribute which determines a successor to this field.

The first three of these are determined by the value attribute in FSL. The successor attribute is the means by which TPS implements the FSL enter and exiting constructs. Normally, the successor of a field is the field immediately following in the FSL source. However, depending on the enter or exiting conditions, the successor of a field may be a later field, indicating that the fields skipped over are irrelevant. The content of an irrelevant field is always considered to be the null string, and is displayed as such, regardless of other attributes.

4.2. Creating a form instance

TPS creates a form instance by evaluating attributes until every field is either irrelevant or has a valid content which has been displayed. There is no need to evaluate the attributes in any order, although, in practice, there is a standard order in which fields are usually considered.

An attribute may be given by an expression or some other means and may thus depend on user input, the content of other fields or information from the database. Every time an attribute is evaluated, a record is kept of any such dependencies. An event is the alteration of data on which an attribute depends, so an event will trigger the re-evaluation of any attributes which have had their dependencies altered. The evaluation of an attribute will then have other effects: altering the display attribute will cause a field to be re-displayed on the screen; altering the validity attribute will affect the validity of a field, and possibly generate an error message; altering the content attribute will give the corresponding field a new content and altering the successor attribute may make certain fields relevant or irrelevant.

The algorithm for completing a form may be very simply given as follows:

```
while the form is not complete
   wait for an event
   while there is an attribute which requires re-evaluation
      choose such an attribute
      re-evaluate it
      depending on the attribute type:
      content
         give the field a new content
      validity
         make the field valid
         or generate an error message
      display
         display the field
      successor
         check the relevancy of following fields
```

The algorithm described is non-deterministic, as it is not specified in what order attributes are to be re-evaluated. There is, however, a natural order for evaluating attributes; they are ordered first by fields, in the same order as given in the FSL source, and are ordered within fields in the order given in the algorithm above. The events will normally be user input to fields in the order given in the FSL source. However, the user is not constrained to that order. It is possible to select fields at random in the form and enter or re-enter data.

4.3. Filing a form instance

The database on which TPS operates has a very simple hierarchical structure; it is viewed as a set of files, where each file is a set of records. A form has associated with it a file identifier. When a completed form instance is filed, TPS creates a record consisting of all the field contents and files it in the identified file. The record identifier is the content of the key field.

When filing a record, TPS must initiate the updates of other records. The implementation of updates in the TPS abstract machine is by a more procedural paradigm than the data collection and validation in form creation, for efficiency reasons. The filing of a document does not generate events but rather has associated with it the execution of a procedure which performs all the necessary updates. In FSL, the update construct is associated with the field whose value is updated rather than the field(s) which causes the update - a feature which reflects the non-procedural nature of the language. The FSL compiler must therefore export an update directive to the form which initiates the update.

5. Transactions in a multi user environment

Each form defines a class of transactions on the database. Even though TPS allows several users to create form instances simultaneously, the designer of the forms may assume that all transactions are atomic. TPS ensures that the total effect of simultaneous transactions is as if they were totally ordered in time. The usual means of providing this

integrity were deemed unsuitable for TPS. Normally, a transaction causes locks to be placed on the database in such a way that other transactions are prohibited from 'interfering' with it [8,10].

This scheme may be seen to be unsuitable for TPS. Consider the case where a transaction reads some datum from the database with the intention of modifying it and writing it back. Such a situation occurs with the form shown in appendix A, where the invoice is used to update the stock levels (using the 'Qty' fields) of product records. The stock level will be read to validate the 'Qty' field and ensure that there is sufficient stock available to complete the invoice. Here, a lock must be placed on the quantity datum which prevents any other invoice transaction from reading it until the original transaction writes back the altered value when the invoice is filed. This would cause unacceptable delays in a real time retail environment with certain heavily used products.

Given the data driven nature of form instance construction, there is a natural way of providing database consistency which uses no locks while transaction (or form instance) construction is taking place. This method has more in common with that of Kung et. al. [13], although they do not provide the interactive modification which we use.

In TPS, a transaction has two distinct phases. During the read phase, no alterations may be made to the database. When data is read during this phase, no locks are associated with it but a tag is provided. If another transaction alters that data, then the tag is read, and a message sent back to the transaction which performed the initial read, informing it that that data is no longer valid and the particular operation must be performed again. During the write phase, a transaction is unrestricted and must operate on valid data. The only lock is a total database lock which allows only one transaction at a time in a write phase. This lock is acceptable in the TPS system since the write phase, which corresponds to filing a form instance, is completely automated and may be performed by a background process without user interaction. By contrast, the read phase, which corresponds to creating a form instance, may take an arbitrarily long period of time while the user enters data. During this time, no locks are created or considered.

5.1. The algorithm

When TPS evaluates an attribute that reads data from the database, a note is made of the dependency in the same way as for dependencies on other fields and labelled with the tag associated with the read operation. If that data is altered, the database server sends a message to the form interpreter with the tag and that attribute is queued for re-evaluation in exactly the same way as when the user provides new data or a field which the attribute uses is altered.

The effect for the user is that the form in use is dynamically updated to reflect the current state of the database. In the example given above, simultaneous invoice transactions will use the same stock level to validate the quantity field for a particular product. When an invoice is filed, all other invoices have the quantity field re-validated immediately and those invoices which become invalid have an

error message produced on the screen.

6. Conclusion

Given the current trends in business data processing towards end-user programming, it is our belief that the FSL-TPS combination provides a powerful yet easy-to-use facility for the solution of many common office problems and one that will significantly reduce the cost of producing such software. In a prototype version of TPS, a complete data processing system covering financial, stock and staff control was implemented as 63 forms written in FDL [19] by an analyst with very little computer experience. The whole system required only a few man-months of effort to construct, once the systems analysis had been completed.

Future developments of the system include enhancements to TPS to provide a mail facility and the integration of an interactive forms design editor. This will significantly increase the level of automation and make available a very powerful tool to the end-user.

REFERENCES

[1] BALBIN I., POOLE P.C, "A Language for Specifying Forms", Proceedings of the Australian Computer Science Conference (6), Sydney, Australia, (February, 1983).

[2] BALBIN I., "A Users Guide to FSL" Technical Report, University of Melbourne, (to appear April 1983).

[3] BERKOWITZ B.T., "Design of a Language for Coding Data-intensive Applications Systems", MSc thesis,MIT, (June 1980).

[4] DATE C.J., "An Introduction to Database Systems", (3rd edition, Addison Wesley).

[5] DE JONG S.P., "The System For Business Automation (SBA): A Unified Application Development System", Information Processing 80, pp.469-474 (1980).

[6] DENIL N.J., "A Business Language", IBM J. Res. Develop., Vol.24 (6) (November 1980).

[7] ELLIS C.A., NUTT G.J., "Office Information Systems and Computer Science", Computing Surveys, Vol.12 (1) pp.27-60 (1980).

[8] ESWARAN K.P., GRAY J.N., LORIE R.A., TRAIGER I.L., "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol.19 (11) pp.624-633 (1976).

[9] FERRANS J.C, "SEDL - A Language for Specifying Integrity Constraints on Office Forms.", Proceedings SIGOA Conference on Office Information Systems, Vol.3 (1,2) pp.123-130 (June 21-23,1982).

[10] J.N. GRAY, R.A. LORIE, G.R. PUTZOLU, I.L. TRAIGER., "Granularity of Locks and Degrees of Consistency in a Shared Data Base", Modelling in Data Base Management Systems, G.M. Nijssen (Ed), pp.365-394 (1976).

[11] HAMMER M. HOWE W.G., KRUSKAL V.J., WLADAWSKY I., "A Very High Level Programming Language for Data Processing Applications", CACM, Vol.20 (11) (1977).

[12] HAMMER M., KUNIN J.S, "Design Principles of an Office Specification Language", AFIPS Conference Proceedings., pp.541-547 (1980).

[13] KUNG H.T., ROBINSON J.T., "On Optimistic Methods for Concurrency Control.", ACM Transactions on Database Systems, Vol.6 (2) pp.213-226 (1981).

[14] KUO H.C., LI C.H., RAMANTHAN J., "A Form-based Approach To Human Engineering Methodologies", Proceedings of the 6th International Conference On Software Engineering, pp.254-263 (1982).

[15] LEBENSOLD J., RADHAKRISHNAN T., JAWORSKI W.M., "A Modelling Tool for Office Information Systems", Proceedings SIGOA Conference on Office Information Systems, Vol.3 (1,2) pp.141-152. (June 21-23,1982).

[16] LEFKOWITZ H.C. et al, "A Status Report on the Activities of the CODASYL End User Facilities Committee (EUFC)", ACM SIGMOD RECORD, Vol.10 (2-3) (August 1979).

[17] LUM V.Y., CHOY D.M., SHU N.C, "OPAS: An office procedure automation system", IBM Systems Journal, Vol.21 (3) (1982).

[18] MILLER P.B., TETELBAUM S., KINCADE N.W., "BUSINESS - An End-User Oriented Application Development Language", ACM SIGMOD Record, Vol.12 (1) pp.38-69 (October 1981).

[19] POOLE P.C., HOLLIER W.E., "A Forms.Description Language", Proceedings of the Australian Computer Science Conference (4), St. Lucia, Australia., Vol.3 (1B) pp.143-153 (May 1981).

[20] ROBINSON M.A, "A Review of Data Base Query Languages", The Australian Computer Journal, Vol.13 (4) pp.143-159 (November 1981).

[21] SHU N.C., LUM V.Y., TUNG F.C., CHANG C.L., "Specification of Forms Processing and Business Procedures for Office Automation", Report RJ3040, IBM Research Laboratory, San Jose, California., (September 1981).

[22] STUART C.J, "Transaction Processing Using Forms in a Multi User Environment", Technical Report (to appear), University of Melbourne, (1983).

[23] THOMAS J.C., GOULD J.D., "A Psychological Study of Query By Example", AFIPS Conference Proceedings, pp.439-445 (1975).

[24] TSICHRITZIS D., "A Form Manipulation System", Technical Report CSRG-101, University of Toronto, (May, 1979).

[25] TSICHRITZIS D., "OFS: An Integrated Form Management System", Proceedings 1980 Conference on VLDB, Montreal, Canada, pp.161-166 (1980).

[26] TSICHRITZIS D., "Form Management", CACM, Vol.25 (7) pp.453-478 (July 1982).

[27] ZLOOF M.M, "Query By Example", NCC (AFIPS) 1975, pp.431-437 (1975).

[28] ZLOOF M.M., "QBE/OBE: A Language for Office and Business Automation", Computer, pp.13-22 (May 1981).

[29] ZLOOF M.M., "Office-by-Example: A business language that unifies data and word processing and electronic mail", IBM Systems Journal, Vol.21 (3) (1982).

APPENDIX A

An Example Form

REGULAR INVOICE

Invoice No. _____ 1 Account No _____ Name _____
Address _____
_____
_____

2 Con/Res ___ Available Credit _____

3 Exempt _____ 4 Reg. No. _____ 5 Order No. _____ 6 Rep. No. ___
7 Settlement Discount [y/n] _ 8 Job No. ____ 9 Non-std y/n _

	Product		Description	Qty	Tax	Unit Price	Discount	Value
10	_____	11	_____	12 ___	13 _	14 _____	15 _____	16 _____
17	_____	18	_____	19 ___	20 _	21 _____	22 _____	23 _____
24	_____	25	_____	26 ___	27 _	28 _____	29 _____	30 _____
31	_____	32	_____	33 ___	34 _	35 _____	36 _____	37 _____
38	_____	39	_____	40 ___	41 _	42 _____	43 _____	44 _____
45	_____	46	_____	47 ___	48 _	49 _____	50 _____	51 _____

Sales tax on $_____ _____

52 less purchases second hands - _____
53 superseded components - _____

54 Instructions _____ TOTAL $_____

print, clear, delete, exit or amend by field __ blemishes:enter b as discount

A Technique To Identify Implicit Information Associated With Modified Code

by

John A. Stankovic
413-5450720

Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, Mass. 01003

Abstract

This paper addresses two of the most difficult problems related to the modification of large complex systems. The first problem is the (unknown) discrepancy that sometimes exists between the specifications and the code itself. A MAP program which can eliminate this problem is described. The MAP program is but one of a set of tools for system modification that are briefly presented here. The second problem is referred to as the 'implicit information' problem and arises in a number of ways. For example, a programmer making a change to existing code can cause complicated interactions in the system causing insidious errors and violations of specifications. One reason for this is that certain specifications may be implemented implicitly. A technique to solve this problem and automatically identify 'implicit information' is proposed. Specific examples of identifying implicit information, using PSL as the specification language and ADA as the coding language, are presented in the paper.

s work was partly supported by the Naval Underwater Systems Center, port, R.I. under contract N000140-81-M-MY61.

1.0 INTRODUCTION

Large complex software systems are constantly in a state of modification. By modification is meant the process of changing requirements, specifications, design, or code due to required functional updates, performance enhancements, or detected errors. In other words, large complex systems are usually in all phases of the life cycle simultaneously with continual modifications needed for a variety of reasons. The modification task is typically very time consuming, costly, and error prone. It is necessary to provide designers and programmers with tools [1,2,4,10,12,13,14,17] to aid this complex modification process. This paper describes a proposed set of modification tools that deal with difficult modification problems. We refer to these problems as the 'specifications to code mapping' problem and the 'implicit information' problem.

In section 2 four proposed modification tools and their relationships are described. These tools are meant to deal specifically with the two modification problems just mentioned. In sections 3 and 4 respectively, each of these two problems is discussed more fully by providing specific examples using PSL as the specification language and ADA as the programming language. For this short paper it is assumed that the reader is somewhat familiar with PSL and ADA. Section 5 summarizes the results.

2.0 MODIFICATION TOOLS

Tools such as PSL/PSA [18,19] support the continual change that occurs in a complex system during the first three phases of the life cycle (requirements analysis, specification, and design). During these stages, the system is described in a meta-language, PSL. This description is stored in a database in computer processible form. Call this the PSL structure map. Such a structure map contains all specified entities and their relationships. See Figure 1 as an example. Using PSA one can then perform a number of analyses on the structure map. When a change is necessary, the PSL description is modified and automatic re-analysis via PSA is possible. However, once the coding begins there is a gap between what is described by PSL and what is implemented.

It is possible to extend the general philosophy of PSL/PSA to the last three phases of the life cycle (coding, testing, and maintenance). To do this we have designed a modification information gathering (MIG) tool, analogous to PSL, that extracts pertinent information (e.g., control and data flow) from the actual code and stores it in computer processible form. We call this a MIG structure map. A modification information analysis (MIA) reports tool, similar to PSA, has also been designed and operates on the MIG structure map. Both the MIG structure map and the output of MIA can be displayed on a graphics device. The tool that provides the display capability is called the modification information display (MID) tool. The display uses a sophisticated computer graphics terminal. Finally, another

program called MAP, is then needed to provide a two-way mapping between the specification (the PSL structure map) and the code (the MIG structure map). This mapping closes the gap between design descriptions and actual code and is an invaluable aid to the modification process as this paper describes. Figure 2 shows the relationships between the PSL, MIG, MIA, MID and MAP tools. In the remainder of this section we provide more detail on the proposed MIG, MIA and MID tools. The MAP tool is treated in section 3.

2.1 MIG

The MIG tool runs on actual code (one module at a time) and produces a MIG structure map that is used by the MID and MAP tools. All information about the actual code is contained in the MIG structure map. To provide the flavor of this structure map this section gives a brief description of some of the information in the MIG structure map (Figure 3). Other information that is kept in the structure map is not shown either because it is irrelevant to the subsequent discussions or because it cannot be drawn conveniently in the Figure.

For each program unit (an ADA procedure, function, package or task) four segments are generated. In addition, a global symbol table is maintained for the entire system and it appears only with the main procedure. An expanded main procedure is shown in Figure 3. This information along with information from other units may be built up over multiple compilation units. The four segments are:

1. program unit information segment,

2. formal parameters segment,

3. declarations segment, and

4. statement structure segment.

The program unit information segment contains the program unit name, type (procedure, function, package, or task), the nested level, all references to this unit, all external references from this unit including those references back into the specifications, whether it is recursive, and pointers to any lower level procedures (not shown in the Figure), etc.

The formal parameters segment contains a complete description of the formal parameters including their name, data type and whether the parameters are in, out, or inout parameters.

The declaration segment contains all variable names declared or used in this unit, the scope of those variables not declared locally, all references to a given variable (local or non-local), the type of reference (read-write), pointers to nested definitions of which this variable is a part, nested program unit declarations which can in turn have further program unit declarations, etc.

The statement structure segment contains the statement type (if, case, assignment, etc.), any nested statements, the statement's position in the control flow (via pointers), a copy of the statement itself, etc.

In summary, the MIG structure map contains all the information about the actual code as well as pointers back to the specifications. These pointers are inserted by the MAP program (see section 3).

2.2 MIA

The MIA reports are very similar to PSA reports except they use the MIG structure map to obtain their information rather than the PSL structure map. We envision a number of MIA reports including listing the overall system module flow, identifying a program that generates a particular output, listing all programs that update a particular file, show all programs called by or calling a particular program, printing all data items that a program uses, print all tasks that can potentially be running in parallel, given an ADA entity then print all PSL entities related to it, etc. Obviously there is a very large set of possible MIA reports that can operate at various levels of detail.

2.3 MID

The purpose of MID is to graphically display program and specification information so as to enhance the modification of code. The display consists of multiple windows, multiple levels of detail, and multiple colors. Each window might contain menus, specification information, control flow information, data flow information, and, in general, output from any of the modification tools . It is also possible to display performance and testing information [14,17] but this is not treated at this time.

Mapping information between specification and code and vice versa is also part of the display and is used when actually performing modifications.

Figure 4 shows an example of the control flow display and Figures 5 and 6 are examples of the data flow display. Using displays such as these a user finds the code to be modified and then is automatically directed to all explicitly related entities (that is, those entities in the modified code's control and data flow), as well as to all related specifications including implicit specifications. It is the user's responsibility to determine how the modified code affects each related entity. The important thing though is that all related entities are identified.

In the remainder of this paper we briefly describe the results of our study on the the two-way mapping between PSL and ADA. The first part of our study is presented in Section 3 and sets the stage for the specific discussion of the implicit information problem and its solution. This latter description is also related to the mapping and is itself divided into two parts. Section 4.1 identifies the implicit information that can be found in PSL objects, and Section 4.2 does the same for PSL relationships. Specific examples are given.

3.0 THE PSL/ADA STUDY: THE MAP

A PSL/ADA mapping is a two-way mapping between specifications and code and is a central entity in the modification process. The PSL specification language was chosen for our study because it is in wide use and the information in which it deals is typical of most specification languages. Mapping the specifications to code is language dependent. ADA was chosen for the language because of its potential wide use and because of its facilities for use on large complex systems.

The PSL -> ADA mapping is accomplished by the MAP program partly automatically and partly with human interaction. The function of the MAP program is to determine all the 1-1, 1-many, many-1 and many-many mappings between specifications and code. The automatic part of the MAP works by requiring two naming conventions: (1) any PSL entity (any legal construct) implemented in some way in ADA should use the same name, and (2) if a PSL entity is implemented by 'n' ADA entities one of the 'n' ADA entities should bear the same name as the PSL entity and this name should be treated as generic with the 'n-1' entity names composed of the generic part and a specific part. For those code entities in which these rules were followed, it is possible to automatically determine 1-1 and 1-many mappings.

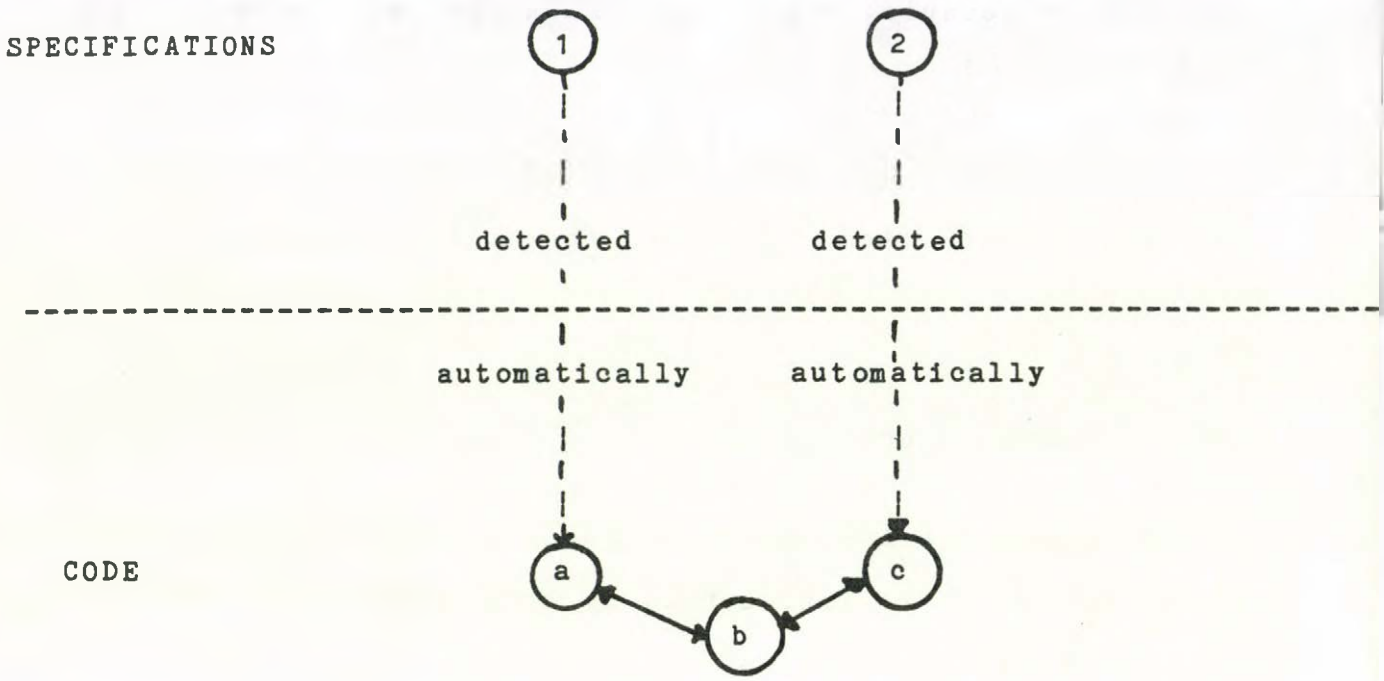
After all automatically identifiable mappings are made then the MAP program presents specification and code entities (one by one) that have not yet been mapped. The programmer (using MID) manually identifies the mappings for these entities. After this

step all remaining 1-1, 1-many and many-1 mappings will be identified because every specification and code entity is treated.

The many-many mappings are more difficult to identify, but they can be identified. Consider the following cases.

CASE 1: Common Subroutines

The picture below shows two specification entities '1' and '2' in which '1' is implemented by code entities 'a' and 'b' and '2' is implemented by 'b' and 'c'. This is a general form of a many-many mapping. The most common case is when entity 'b' is a subroutine. In this case 'b' is linked to 'a' and 'c' in the code. When the programmer is presented 'b' as an unmapped entity, even if he assigns it just specification '1' (or '2') the link to the other specification is determined at modification time through the explicit link between 'b' and 'c' in the code itself.



CASE 2: GENERIC NAMING

Consider the same case as above except that 'b' is named using 'a' as the generic part. In this case it will automatically be linked to '1' and the link to '2' will go undetected. However, if 'b' is a subroutine as above then the relationship to '2' will be detected at modification time.

CASE 3: 'a', 'b', 'c' DISJOINT.

Consider that there are absolutely no links between 'a', 'b' and 'c'. Hence when 'b' is presented to the programmer he may assign either '1' or '2' without knowing of the other's existence. Furthermore, since 'b' is disjoint from 'a' and 'c' no explicit links in the code can identify the missing specification. However, in this case there will be a link

between '1' and '2' in the specifications and this will be found at modification time. Specifications '1' and '2' must be linked because they need the same entity which is being implemented by 'b'.

Of course, it would be desirable to remove the need for human interaction. This might be accomplished by redesigning both the specification language and the programming language to be a better match. Such a discussion is not the subject of this paper. See [3,7,8,11].

As the mapping between PSL -> ADA occurs MAP also inserts backward pointers, automatically providing the ADA -> PSL mapping to be used by the MID tool when actually making code modifications. Using the two-way mapping as part of the tool set MIG/MIA/MID/MAP eliminates discrepencies between the specifications and the code [13]. We now briefly discuss the mapping itself in terms of PSL objects and relationships.

3.1 PSL Objects And ADA

Objects and relationships are the primitives used in PSL to write specifications. There are 28 types of objects and 75 types of relationships. A PSL object is anything given a PSL name. Each object is given a unique name so that it can be identified each time it occurs in the system description. Consequently, all occurrences can be collected and analyzed. We extend this by saying that any PSL object implemented in some way in ADA should use the same name or generic variation, if possible. This

extends the recognition capability to the actual code as described above. Some specification entities (objects or relationships) map nicely to code, e.g., a PSL PROCESS object is either a Procedure, Task, Function or Package in ADA.

Other PSL objects have a wide range of options of how they might be implemented, e.g., a PSL ENTITY object may appear in the code as a constant, an element of an array or record, or part of an enumerated type, etc.

On the other hand, not all PSL objects will be found in ADA code, e.g., the Project Management objects (MAILBOX and PROBLEM DEFINER), the Organization object (INTERFACE) and the System Architecture objects (PROCESSOR, RESOURCE, UNIT, and RESOURCE USAGE PARAMETER). This is because such objects are not implemented in the code. Still other PSL objects provide implicit information about the system. Examples of this are presented in section 4. Hence, the mapping between PSL objects and ADA code is certainly not a clean mapping. Table 1 summarizes the allowable PSL objects -> ADA mappings.

3.2 PSL Relationships And ADA

In mapping PSL relationships to ADA code one finds the same general issues as discussed above. First, some PSL relationships map nicely to ADA code. For example, a PSL UPDATES relationship between a PROCESS and a SET is any write reference to the ADA entity representing that set. Second, some PSL relationships are not found in the code. For example, any PSL relationship that

refers to objects not found in the code is also not found in the code (e.g., GENERATES/GENERATED BY relationships when applied to INTERFACE objects, and all CONSUMES/CONSUMED BY, PERFORM/PERFORMED BY, MEASURES/MEASURED BY relationships applied to any objects because these relationships only apply to objects not found in the code). In yet other cases, some PSL relationships have a wide range of options of how they might be implemented, e.g., the USES relationship can be implemented as any reference to an object, the CAUSES relationship might be implemented as an interrupt, task initiation or termination, or the ADA RAISE statement, etc., and the CONSISTS OF relationship might be implemented as almost any structured data declaration. Finally, as for PSL objects, other PSL relationships often impart implicit information about the system (see section 4.2). The complete description of the mapping of PSL relationships to ADA code can be found in [13]. A sample is presented in Table 2. The entire mapping is too large and complex for inclusion in this paper.

In summary, the MAP program has knowledge of the PSL and MIG structure maps. Using common and generic names for entities many links are determined automatically by MAP. Each specification not linked to code is presented to the user for manual linking or an assessment that it is not intended to be in the code. Hence every specification is treated and therefore discrepencies can be identified. After modifications, only modified specifications and code need be (re)-linked. Certain types of legality checks on the mappings can be performed by MAP using tables such as

Tables 1 and 2.

4.0 IMPLICIT INFORMATION TECHNIQUE

A major problem with the modification process occurs when one attempts to actually modify the code. Often the changed or added code itself can be tested and all local bugs removed. However, there is always a potential for introducing errors beyond those that exist in the new code itself. What is needed is some mechanism for determining the effect of this modified code on the rest of the system. Exacerbating the problem is the fact that the effect may be felt in both explicit and implicit ways. Explicit interactions between the modified code and the rest of the system occur via calls and direct or indirect data references. Explicit interactions in the code can be traced in a systematic and recursive (although possibly tedious) fashion using a tool such as MID, STRUCT [14], or ESTRUCT [12]. A programmer or designer can then determine if any of these explicit interacting entities are affected by the modification.

How does a programmer find problems caused by modified code when the problems are due to implicit relationships that exist in the system? By implicit relationship is meant any relationship which is not directly implemented by code or data itself. Examples are provided later in this section. It is our hypothesis that a significant number of implicit relationships can be identified by using MID and MAP. Remember, specifications often contain descriptions of objects and relationships that are only implicitly found in the code. Using the technique and tools

presented here, it is possible to automatically detect any such implicit relationship. That is, whenever modifying code, the backward mapping to the specifications is used to identify both explicit and implicit specification information associated with the code entities being modified. Note, any missing implicit relationships in the specifications are an oversight and are themselves errors.

The technique to identify implicit information associated with modified code begins with the programmer scanning (scrolling) through the system code using MID. The code is displayed in flowchart form. Modifications to the code are then made through MID. All explicit code entities potentially affected by this change can then be automatically and systematically identified using the MIG database. For each of these potentially affected code entities, the MAP program also identifies all specification entities related to that particular code entity, including those specification entities not directly implemented in the code. That is, some specification entities may contain implicit information about the system. It is the programmer's responsiblity to determine if any of the related entities must change. If there is a subsequent programming change required, then this secondary change is treated in the same manner as the first modification, and so on in a recursive fashion. The process completes when all interactions (explicit and implicit) are identified, studied and modified, if necessary. In subsections 4.1 and 4.2 some examples of implicit information contained in specifications are provided. Then in subsection 4.3

some additional remarks about implicit information in large systems are made.

4.1 Examples From PSL Objects

RELATIONS, ATTRIBUTES, and CLASSIFICATION are the only three PSL objects that are potential sources of implicit information that affect the code itself.

RELATIONS describe the logical connections between entities. RELATIONS may map to specific ADA code, e.g., an access type, or actual code that performs some type of consistency check. In other cases, the RELATION may not be explicitly implemented in the code. In this case modifying any of the entities to which this RELATION refers may invalidate the RELATION unbeknownst to the programmer. Our technique maps back to the specification, automatically identifying all associated RELATIONS of a modified entity. In fact, all associated PSL objects and relationships are found, not just RELATIONS. The programmer can then check that this implicit RELATION requirement, as well as any other requirements, still holds.

More specific examples are now given. Object A (copy 1 of some data structure) must always be consistent with Object B (copy 2 of the same data structure). Assume that this is a requirement defined as a RELATION object in PSL. Also assume that the original code meets this requirement by having the two data structures updated together or not at all. A subsequent modification to the code may erroneously permit the two updates

to occur at different times. However, with our scheme, changing either Object A or B, or code that references A or B, will automatically identify the RELATION object which states that both objects must be updated together. The programmer seeing the requirement, notices (possibly with the use of additional tools) that the modification fails to uphold the requirement, so it is in error.

The same process of checking all related information about modified entities through the MID graphics tool applies to all of the following examples and therefore we do not repeat this fact.

As another example, suppose PROCESS A has ATTRIBUTE terminal, i.e., it cannot call any other PROCESS. Assume that originally this was satisfied by programming PROCESS A as a procedure that contained no call statements. At some later time it is easy to violate this requirement (by adding code that has a call statement) because it is only implicitly satisfied in the code. Again, our mapping would detect such an error. Of course, in some cases it may be decided that changing the requirement is needed.

Object A (data structure A) can be legally accessed by PROCESS B and C only. This requirement can be stated by a PSL CLASSIFICATION object. Any modified code, other than in PROCESS B and C, now accessing Object A is wrong. When checking the correctness of the modified code its data reference to Object A is explicit and the associated implicit CLASSIFICATION information is also found.

4.2 Examples From PSL Relationships

Over half of the PSL relationships can impart implicit information about the system. Five examples are presented in this section.

The relationship ASSERT may require that a particular input buffer hold enough characters to allow double buffering and prevent a loss of characters at all costs. A code modification may be made to reduce the size of the buffer for memory efficiency reasons. However, this reduction may cause instances where characters are lost. Identification of this requirement (which is implicitly implemented by choosing the correct buffer size based on device speeds) at the time of code modification should result in another calculation of minimum buffer size. This would avoid the error.

A PSL FOREACH relationship might specify that the salary field in a file of employee records be encrypted. Code adding new records may erroneously omit the encryption of the salary field. The associated FOREACH relationship would enable the programmer to detect the error.

In PSL the relationship SUBPARTS implies that the entities connected by this relationship are completely homogeneous. Assume a file of employee records called workers. A modification to the code might permit manager records to be part of the file. While conceptually this is fine, it is a violation to the implicit homogeneous requirement.

Another PSL relationship is CARDINALITY. A requirement for a particular system might be that the maximum CARDINALITY of the instantiations of PROCESS A is four. There may be no explicit code in the system guaranteeing this requirement. Yet, because of the system configuration the original designers were sure that no more that four would ever exist. Subsequent modifications could easily lose track of this requirement and allow this limit to be exceeded. In this case any new code allowing a new instantiation of PROCESS A will identify the CARDINALITY requirement. Determining if the requirement is met may still be quite a difficult task.

DERIVATION and PROCEDURE relationships are comment entries about RELATIONS or SETS, and PROCESSES, respectively. Whenever code entities corresponding to RELATIONS, SETS, or PROCESSES are modified the programmer will automatically be given the DERIVATION or PROCEDURE comments for determination of possible implicit information contained therein. A PROCEDURE comment about PROCESS A might state "it was decided to use QUICKSORT as the sorting algorithm because we expect very large lists of highly unsorted elements." A programmer could easily decide to change the system by substituting another sort algorithm from a library not knowing the implicitly stated assumptions about the inputs to the sorting algorithm.

4.3 Remarks On Implicit Information

We believe that the technique of identifying implicit information presented in this paper can have a substantial impact on improving the modification process. The technique helps the programmer or designer avoid many difficult to detect errors. To date, there are no techniques to help detect some of these errors. However, the technique is not a panacea. At times, implicit relationships exist in the system without anyone's knowledge. It seems impossible to automatically identify these relationships. Only through a laborious debugging process are these implicit relationships found, if ever. Once identified, though, they can be added to the specifications and, thereafter, always identified at the proper time during subsequent modifications.

In practice, the most difficult implicit relationships to identify are often related to timing and other real-time issues. This implies that more effort at understanding these issues while writing the specifications, as well as better facilities in specification languages for describing real-time requirements are required. If these two things are done then, in theory, implicit information in real-time issues can also be handled by our technique.

Note that the MAP is the tool which integrates the specification tools with the programming language tools. Such a MAP tool is of utmost importance in developing an integrated software engineering environment. Another approach, currently

being actively researched, is to automatically generate correct code directly from specs (either in one step or incrementally). This would eliminate the need for a MAP program. However, this research has not yet progressed enough to be useful for large complex systems. Until such an occurrence, a MAP program is invaluable. Furthermore, a MAP tool can be implemented with today's technology.

5.0 SUMMARY

Modifying the code of large complex systems is extremely difficult and costly. This is often due to the large number of complex interactions in such systems. Good design methodologies attempt to limit the interactions. Invariably, however, the programmers are unaware of some of the explicit and implicit relationships and requirements that exist, giving rise to errors. This paper reports on a technique to systematically identify related entities in the system, assuming that specifications are written in PSL and that the code is written in ADA. This is accomplished by a set of proposed tools, MIG/MIA/MID/MAP. An important result is that even implicit relationships which are often the cause of the most difficult to detect errors, can be identified by this technique. Of course, all that is needed to extend this technique to other specification and/or programming languages is different MIG/MIA/MID/MAP programs. The tools could also be extended to include code performance evaluation [14][15][16] and testing information.

Note also, that the mapping between complex languages such as PSL and ADA requires some human interaction. Since elimination of the need for human interaction is desirable, this is where continued research is required. Possible solutions include the use of more formal specification and programming languages, and automatically generating code from specifications.

The tools proposed here have not been implemented due to lack of resources. However, we believe that the merit of the main ideas presented here is shown both by a description of how to implement the tools and by specific examples of their use.

6.0 REFERENCES

[1] Adam A., P. Gloess, and J. P. Laurent, "An Interactive Tool For Program Manipulation," Fifth International Conference on Software Engineering, San Diego, California, March 9-12, 1981.

[2] Azuma, M., M. Takahashi, S. Kamuja, and K. Minomura, "Interactive Software Development Tool: ISDT," Fifth International Conference on Software Engineering, San Diego, California, March 9-12, 1981.

[3] Cheheyl, Maureen Harris, Morrie Gasser, George A. Huff, and Jonathan Millen, "Verifying Security," Computing Surveys, Vol. 13, No. 3, Sept. 1981, pp. 279-339.

[4] Felty, James L., and Mark Davis, "SPAR (Source Program Analyzer and Reporter)," IR-215-1, Intermetrics, Inc., January, 1978.

[5] Ichbiah, Jean, et al, Reference Manual For the ADA Programming Language, Proposed Standard Document, United States Department of Defense, July, 1980.

[6] Landwehr, Carl E., "Formal Models for Computer Security," Computing Surveys, Vol. 13, No. 3, Sept. 1981, pp. 247-278.

[7] Levene, A. A. and G. P. Mullery, "An Investigation of RequirementSpecification Languages: Theory and Practice," IEEE Computer, Vol. 15, No. 5, May 1982.

[8] Ludewig, Jochen, "Computer-Aided Specification of Process Control Systems," IEEE Computer, Vol. 15. No. 5, May 1982.

[9] Medina-Mora, R., and P. H. Feiler, "An Incremental Programming Environment," IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, September 1981.

[10] Miller, Ed, Tutorial: Automated Tools for Software Engineering, IEEE Computer Society, November 1979.

[11] Popek, Gerald J., and David Farber, "A Model for Verification of Data Security in Operating Systems," CACM, Vol. 21, No. 9, Sept. 1978, pp. 737-749.

[12] Stankovic, John A., Structured Systems and Their Performance Improvement Through Vertical Migration, UMI Research Press, Ann Arbor, Michigan, 1982.

[13] Stankovic, John A., Software Tools For the Support of System Modification, Final Report, Naval Underwater Systems Center, Newport, Rhode Island, January 1982.

[14] Stankovic, John A., "Good System Structure Features: Their Complexity and Execution Time Cost," IEEE Transactions On Software Engineering, Vol. SE-8, No. 4, pp. 306-318, July 1982.

[15] Stankovic, John A., "Improving System Structure and Its Affect on Vertical Migration," Microprocessing and Microprogramming, Vol. 8, No. 3,4,5, pp. 203-218, December 1981.

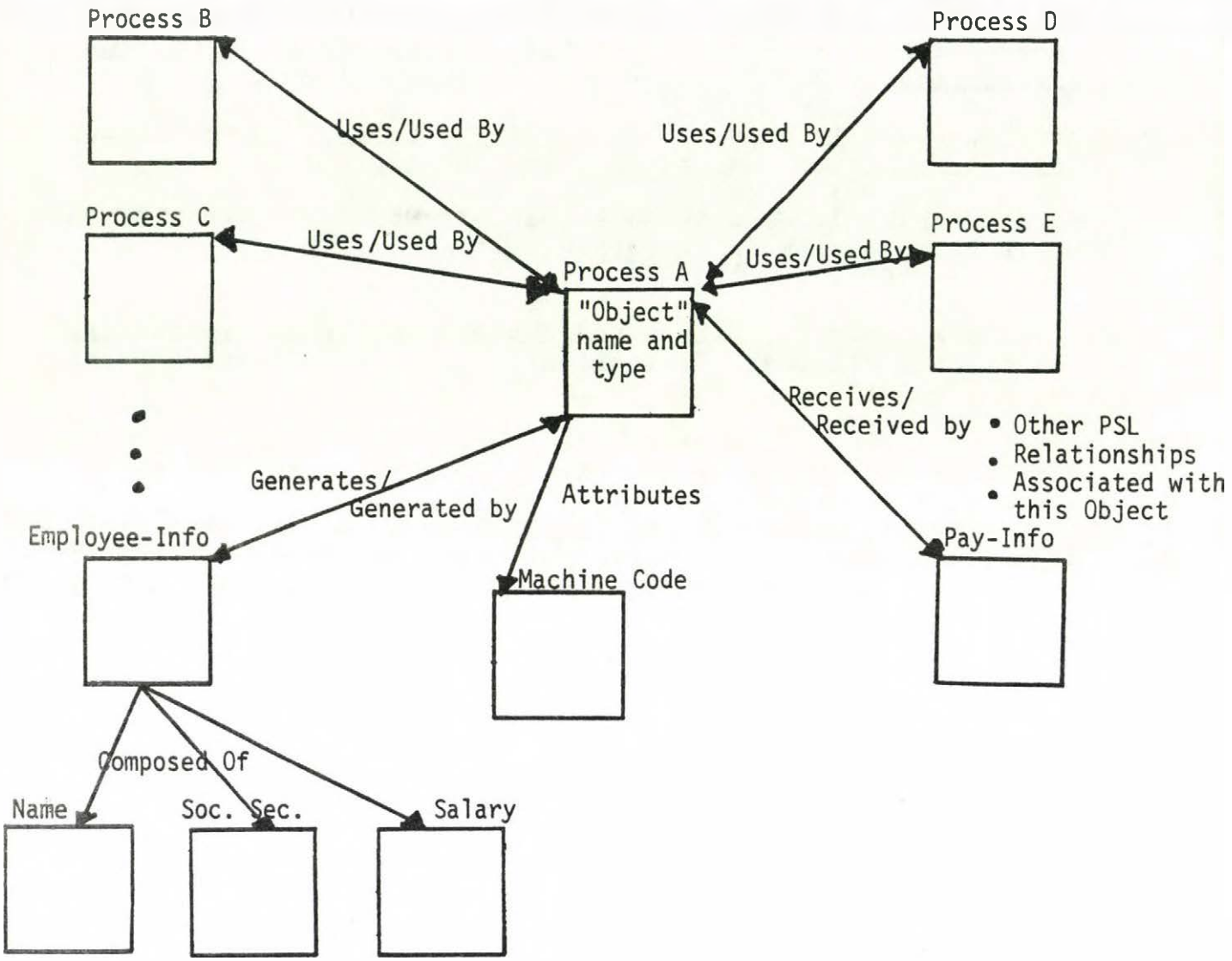
[16] Stankovic, John A., "The Types and Interactions of Vertical Migrations of Functions in a Multi-Level Interpretive System," IEEE Transactions on Computers, Vol. C30, NO. 7, pp. 505-513, July 1981.

[17] Stockenberg, John E. and Andries van Dam, "STRUCT Programming Analysis System," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, December 1975.

[18] Teichrow, D., and E. A. Hershey III, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, SE-3(1) pp. 41-48, 1977.

[19] AN/UYK-7/SHARE-7/URL User's Manual, Part 1, Electronic Systems Division, Department of the Air Force, June 1978.

Figure 1: Hypothesized PSL Structure Map



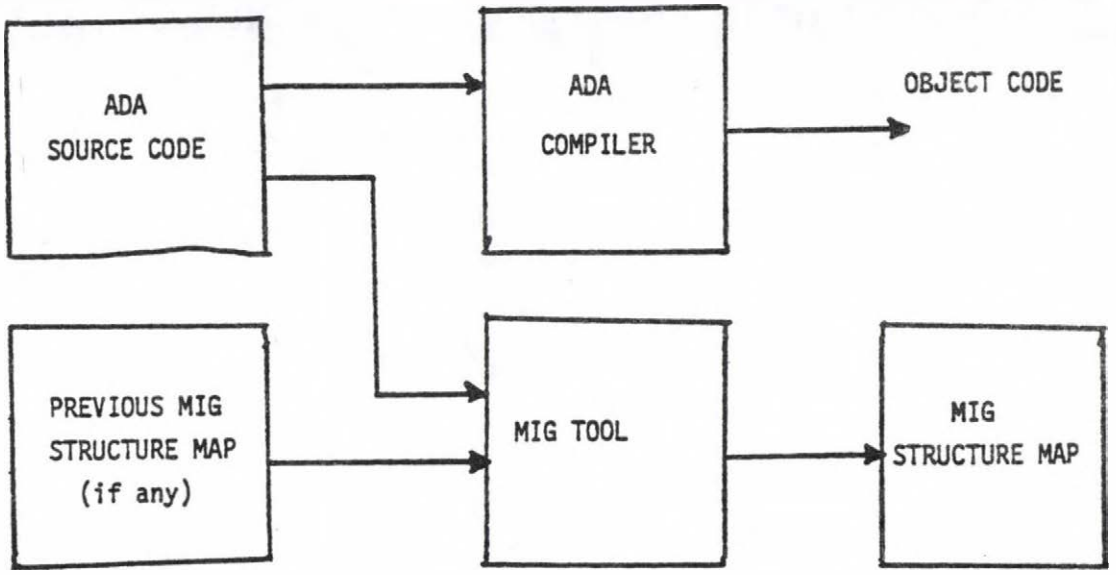
Legend

Box: Represents PSL objects containing their name and type

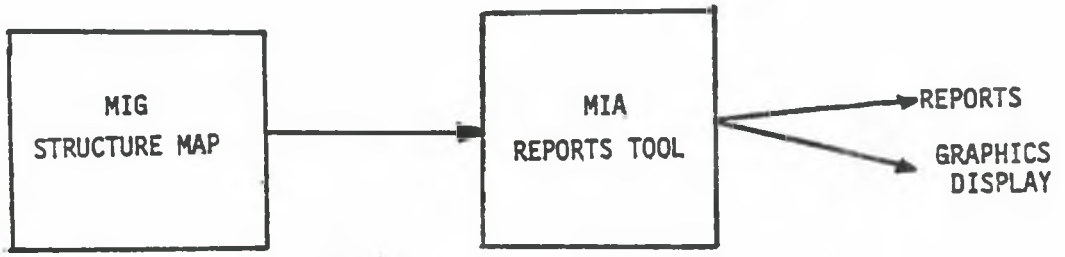
Arcs: Represents PSL relationships

Note: By following links all related entities can be determined.

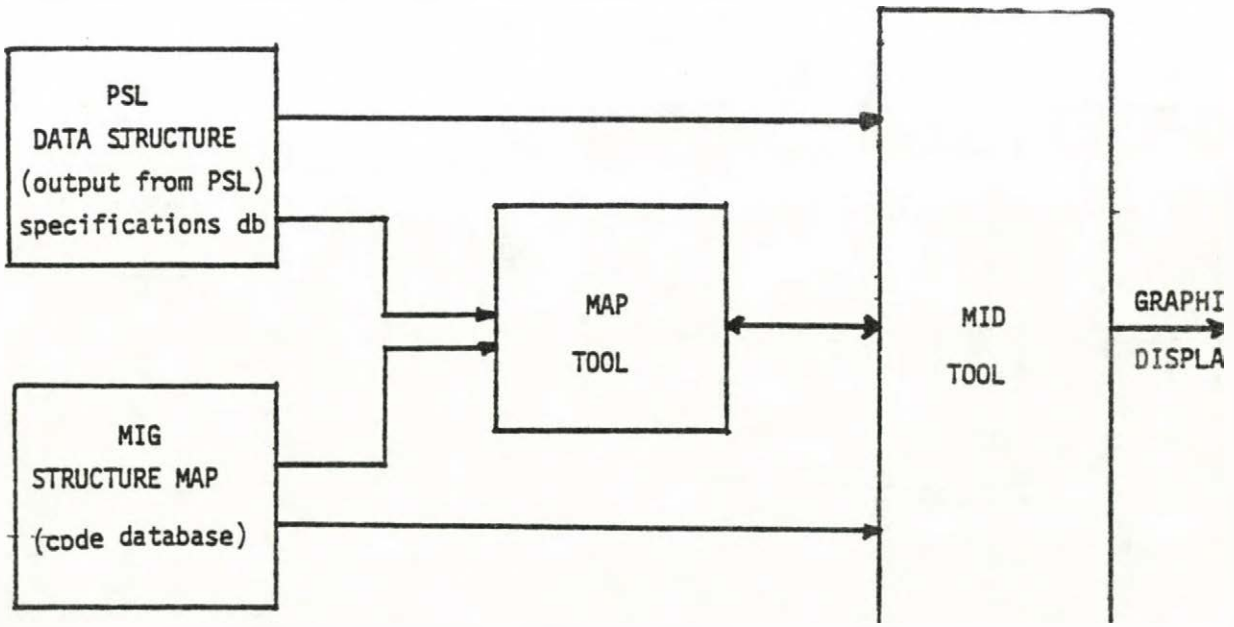
FIGURE 2: MIG/MIA/MID/MAP Tools



(a) MIG



(b) MIA



Note: This can be built up over multiple separate compilations.

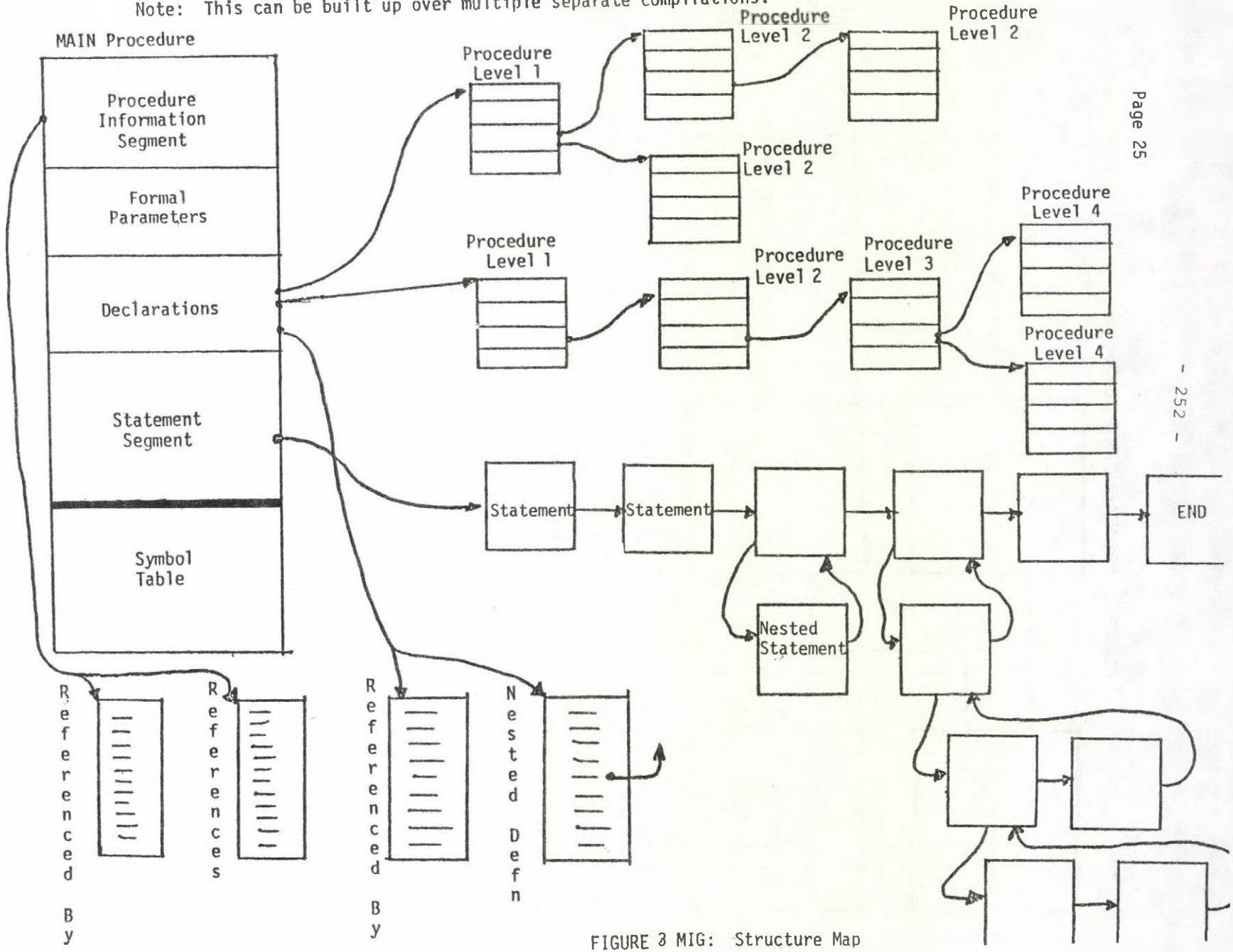


FIGURE 3 MIG: Structure Map

FIGURE 4: Control Flow Display

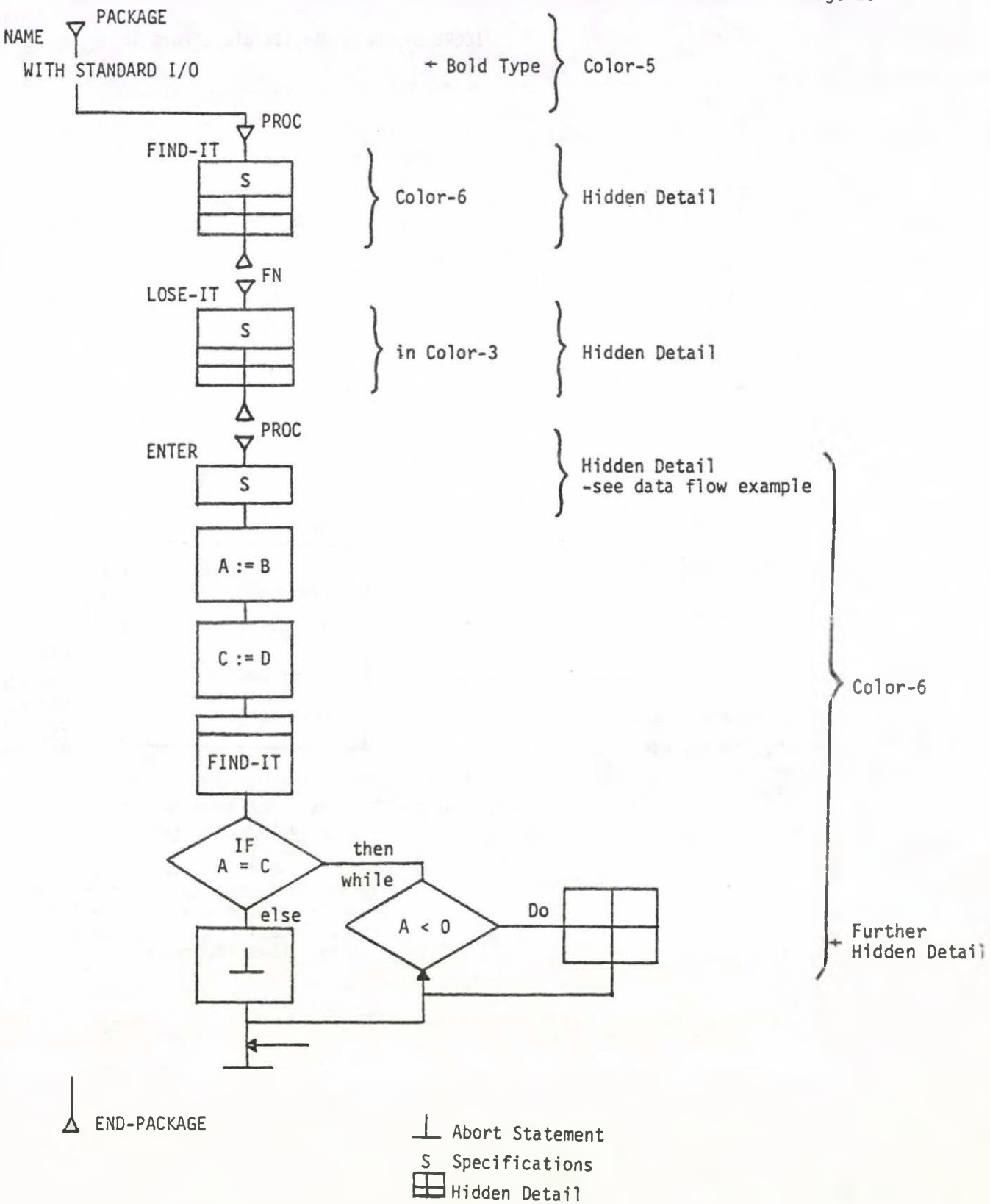
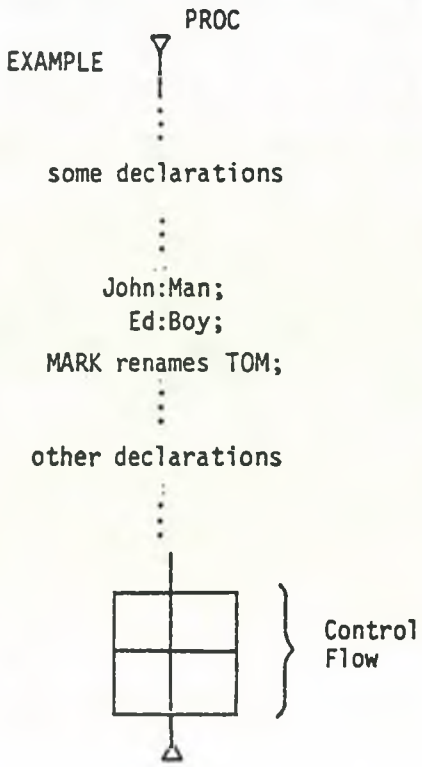
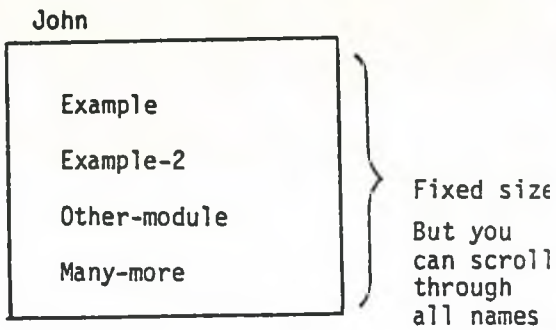


FIGURE 5: Data Flow Display, Part 1



User lightpens name JOHN
and asks for all users



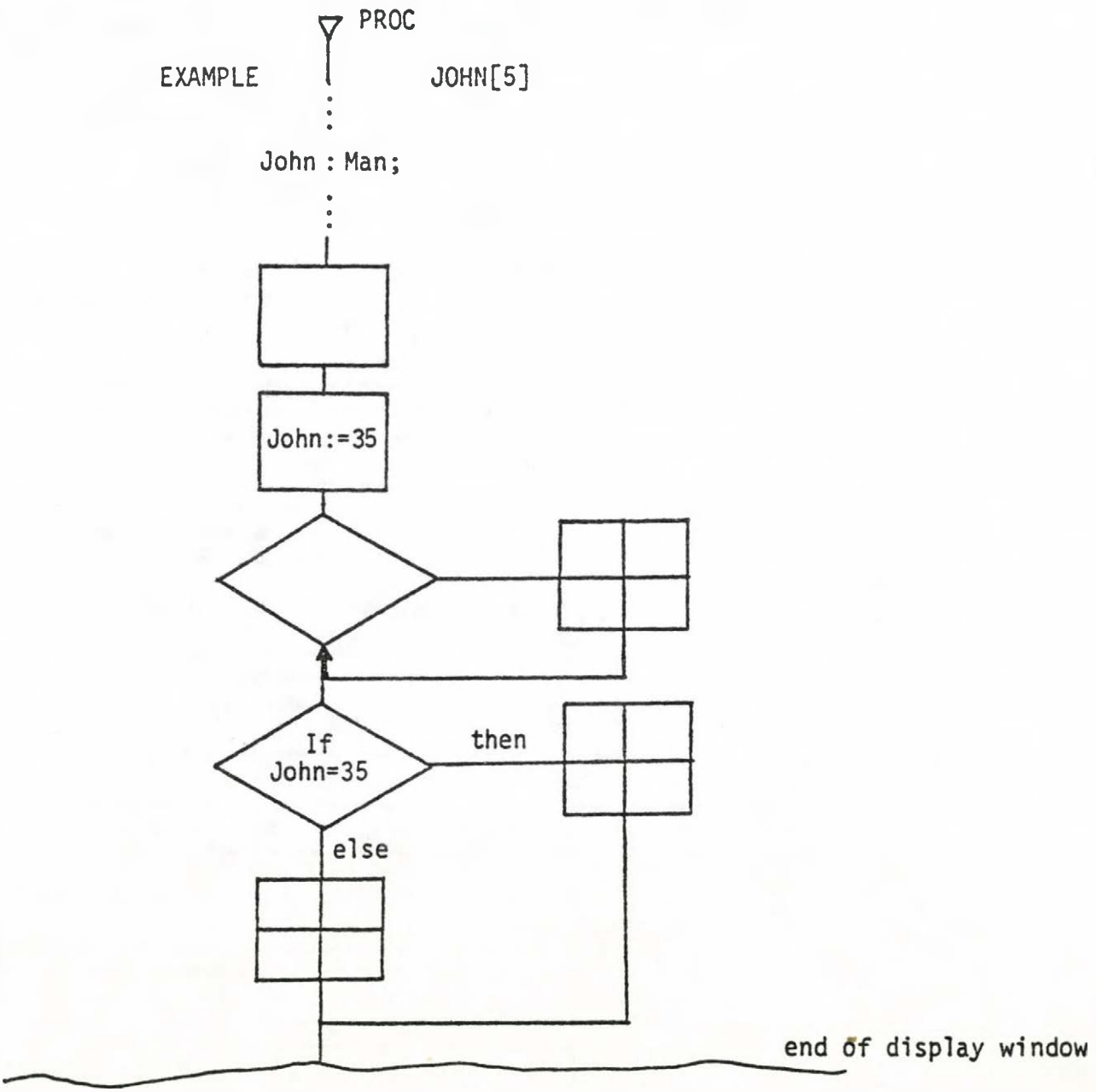
Note: Only users of this particular entity "John" are listed.

Note: This is the first level of detail. By now lightpenning the subprogram name in the above box one gets the actual location(s) doing the referencing. (See Figure 6).

*For this figure we use size of 4 but on the actual display it would be set to a larger number.

FIGURE 6: Data Flow Display, Part 2

Note: Seeing Figure 5 on the screen, suppose a user lightpens the word "Example" in the box below the identifier JOHN. He may get the following display.



*Note: Three references in this module are visible on the display, but the JOHN[5] display indicates a total of 5 are made. Scrolling the control flow is necessary to view the other 2 references.

TABLE 1

PSL OBJECT TYPE	ALLOWABLE ADA CONSTRUCTS
Attribute	All Ada constructs are allowable Ex. - characteristics of types and subtypes, BASE, RANGE, FIRST, LAST, POS, SUCC, PRED, VAL, DIGITS, ARRAY, MANTISSA, EMAX, SMALL, LARGE, EPSILON, LENGTH, etc.
Attribute-Value	Specific values of ADA constructs corresponding to Attributes listed above
Classification	PRIVATE and to some extent visibility rules.
Condition	Boolean data type, exceptions
Element	Fields of a record, elements in an array, constants, variables
Entity	Any ADA data representation, e.g., ARRAYS, CONSTANTS, RECORDS, etc.
Event	TASK, RENDEZVOUS, SELECT, WHEN, DELAY, ENTRY, INTERRUPTS, CALL
Group	TYPES, ARRAYS, RECORDS
Inputs	FILE
Interface	NA
Interval	DELAY or via human interaction
Keyword	NA
Mailbox	NA
Memo	Human interaction and comments
Outputs	FILE
Problem-Definer	NA

TABLE 1 Cont.

PSL OBJECT TYPE	ALLOWABLE ADA CONSTRUCT
Process	PROCEDURE, TASK, FUNCTION, PACKAGE
Processor	NA
Relation	Acess types and human interaction
Resource	NA
Resource-Usage-Parameter	NA
Security	PRIVATE and to some extent visibility rules.
Source	NA
SET	Any compound data structure
Synonym	RENAME
System-Parameter	Constants, representation specs, human interaction
Trace-Key	NA
Unit	NA

TABLE 2: PSL RELATIONSHIPS ⇒ ADA CODE

PSL SYSTEM STRUCTURE RELATIONSHIP

Object	PSL Relationship	Object	ADA Construct
Set	Subset of/subsets	Set	Implicit in Compound Structure Declarations
Process	Utilize/Utilized by	Process	Call statements
Interfaces	Subparts are/part of	Interfaces	NA
Inputs	Subparts are/part of	Inputs	In Declaration for Records of a File
Outputs	Subparts are/part of	Outputs	In Declaration for Records of a File
Processes	Subparts are/part of	Processes	Nested Declarations of Procedures, Functions, Tasks and Packages
Processors	Subparts are/part of	Processors	NA

MICRO-PSL and the Teaching of Systems Analysis and Design

R J Thomas

J A Kirkham

University of Bradford, U.K.

Working Conference

on

Systems Description Methodologies

May 23-27, 1983

Kecskemet, Hungary

IFIP TC2 Programming

MICRO-PSL and the Teaching of Systems Analysis & Design

1. Introduction

Systems Analysis and Design has been taught as an academic discipline at Bradford for the past ten years to both Postgraduate and Undergraduate students. In that time our teaching method has slowly evolved from the traditional approach to Systems Analysis with its well known phases of investigation, analysis, design and implementation. In this approach the analysis consisted of the distillation of the results of the initial investigation into a suitable documented form. This was followed by a design stage which considered each of the required outputs and then derived the inputs, files and processes needed to produce them.

No formal step by step methodology was used in this procedure. The students were expected to acquire the "principles" of analysis and design by applying the various tools that they had been given to a large and diverse set of Case Studies, each of which attempted to simulate a real life situation. The outcome was inevitable. Good students were able to improvise on the outline instructions they had been given and were able to synthesise them into a workable methodology. Poor students were lost. What was needed was a more rigorous approach - a formal step by step methodology which would take them from the initial problem definition to the final physical system specification. (Similar problems of a lack of a formal design methodology were encountered by teachers of computer programming before the advent of Structured Programming).

2. Current Teaching Methodology

Our current teaching is based upon Gane and Sarson's and Page-Jones's Structured Systems Analysis and Design Methodologies (1),(2). Gane and Sarson begin the IPS development cycle by describing the flow of information through an existing system using a logical Data Flow Diagram (DFD). Systems are modelled using only three basic symbols (Figure 1) which represent an EXTERNAL ENTITY (a source or destination of data), a DATA STORE and a PROCESS. Information travels between these three types of object via Data Flow lines, each such line representing a pipe-line along which the data named on the line will flow.

A typical example is shown in Figure 2 where a CUSTOMER sends ORDERS which may be physically contained in a letter, a telephone call or a satellite link to the process PROCESS-ORDERS. The process is something which we are interested in analysing and which will subsequently become the subject of a more detailed DFD. It may be a room full of clerks, a computer program or a combination of both. It uses data from the data stores PRODUCT-DATA and CUSTOMER-DATA to check the validity of the ORDERS and if all is well INVOICES (together with the goods ordered) are sent to the CUSTOMER. At this point in the definition of the data flow within the system, there is no mention of how the various activities are carried out. This clearly differentiates between the logical analysis phase of IPS development and the subsequent physical design and implementation.

continued.....

The analysis continues as PROCESS-ORDERS is expanded to show the main data flows within the process, without showing any error or exception handling. This is called a first level diagram and is designed to show the overall flow or "big picture" of the system. Figure 3 shows PROCESS-ORDERS expanded into a typical level 1 DFD. The processes defined at this level are expanded at the second or lower levels to include the detail of error and exception handling.

Once the DFD's have been constructed in sufficient detail, the data elements used in the system are identified, named and placed in a data dictionary. Figure 4 contains illustrative examples of the five types of forms used to describe each Data Element, Data Structure, Data Flow, Data Store and Process within the system.

Gane and Sarson suggest that the data dictionary could be maintained as a set of index cards with the cards grouped alphabetically within type for ease of access. However it should be clear that maintaining such a dictionary would be a much simpler task if the records were held on a computer. Furthermore the ability to sort, sift and select the data would then be considerably enhanced.

At the end of this analysis the DFD's and data dictionary make up a comprehensive description of the system called the logical functional specification which describes what the system does (or is to do) without any reference to its physical implementation. This clear separation of the specification and analysis of IPS requirements from the subsequent design and physical implementation of a system is a valuable feature of the Gane and Sarson approach.

Furthermore, the hierarchical nature of the DFD's imposes a top down approach to the analysis of information flows, providing a more rigorous methodology in the analysis phase.

At this point, however, the formal methodology disappears. We revert to our earlier approach to designing systems. Using our ingenuity and our experience (where do students get their experience from?) we eventually arrive at a complete specification of the physical system required to solve our problems. This specification consists of the traditional documentation of inputs, outputs, files and processing which we were producing ten years ago.

Clearly our attempt at devising a complete formal methodology for systems analysis and design has not made much progress beyond the specification of requirements and analysis stages of systems development.

continued.....

3. Problems with Current Teaching Methodology

3.1 Our current methodology is a mixture of new ideas (Structured Systems Analysis) and traditional techniques (Input, Output, Database design). It lacks a clearly defined step by step methodology which will take us from the initial problem definition to the final systems specification.

3.2 Gane and Sarson's and Page-Jones' methodologies require the drawing of a large number of DFD's of varying complexity. Drawing and modifying the diagrams is a difficult and time consuming task. With the subsequent computer run diagrams of the physical system to be drawn as well, it is clear that drawing and maintaining the diagrams is a major problem.

The Data Dictionary supporting the DFD's is also produced manually. This is a laborious, error prone procedure and a source of some frustration among the students. Changes to the DFD's mean corresponding changes to the Data Dictionary with the consequent problems of maintaining consistency between the diagrams and the dictionary.

3.3 Student solutions to Case Studies consist of a set of DFD's, a data dictionary and a specification of all inputs, outputs, files and processing required. Checking a single solution manually for completeness and consistency as well as the "quality" of the proposed design is a very difficult task. With groups of 25 students or more it becomes virtually impossible. Something must be done to alleviate this problem.

4. Computer-Aided Systems Analysis - MICRO/PSL

We have been aware for some time that a computer could be used to provide assistance in systems development. In fact in 1977-79 we were using our own systems documentation software package at Bradford on Systems Analysis courses (3).

During 1979 we investigated the possibility of using the ISDOS PSL/PSA system (4) at Bradford but at that time our central computing facilities proved to be insufficient to handle the package. Consequently, we decided to try to develop a much smaller package based upon both the language PSL and the reporting facilities of PSL/PSA.

MICRO-PSL is a software system which has been developed at the University of Bradford with SERC support. The system is modelled on the PSL/PSA mainframe package developed by the ISDOS group at the University of Michigan, USA(4). It consists of a language PSL which is used to describe functional specifications of information systems, together with a program suite which analyses the PSL statements and stores the specification on a database. A Report Package is provided which enables the analyst to check on the consistency and completeness of the specification.

continued....

4.1 Problem Description in PSL

Functional models are described by a series of English like PSL statements which are checked for correct syntax and then stored in a data base. For example the DFD shown in Figure 2 is written in PSL as:-

```
DEFINE INTERFACE CUSTOMER;
            GENERATES ORDERS;
            RECEIVES INVOICES;
DEFINE PROCESS PROCESS-ORDERS;
            RECEIVES ORDERS;
            DERIVES INVOICES;
            USES PRODUCT-DATA,
                 CUSTOMER-DATA;
DEFINE SET CUSTOMER-DATA;
            CONSISTS OF CREDIT-STATUS;
            USED BY PROCESS-ORDERS;
DEFINE SET PRODUCT-DATA;
            CONSISTS OF PRODUCT-DETAILS;
            USED BY PROCESS-ORDERS;
DEFINE INPUT ORDERS;
            GENERATED BY CUSTOMER;
            USED BY PROCESS-ORDERS;
DEFINE ELEMENT CREDIT-STATUS;
            CONTAINED IN CUSTOMER-DATA;
DEFINE OUTPUT INVOICES;
            RECEIVED BY CUSTOMER;
            DERIVED BY PROCESS-ORDERS;
```

These PSL statements are input to MICRO-PSL which checks and stores them in a data base. The first and subsequent levels of the model are developed in a similar manner and stored in the data base.

4.2 Entry of PSL Description

PSL statements are entered into MICRO-PSL by the user of the system via a VDU. The accuracy of each statement is initially checked for correct syntax and then for consistency against any PSL statements already entered into the MICRO-PSL database. The parsed form of the statement is then displayed on the screen to allow for immediate correction, should this be necessary.

As an illustration when the statement:-

```
USES ORDER TO UPDATE PRODUCT-DATA;
```

has been entered, this would be analysed and displayed on the screen as:-

```
USES
ORDER                          UNDEFINED
TO
UPDATE
PRODUCT-DATA                   SET
;
Accept, Reject, Edit ?
```

continued....

The statement has been analysed into its constituent parts and the object ORDER is reported to be UNDEFINED. Although the statement is syntactically correct and would be stored on the database if Accepted, it may be logically incorrect.

The UNDEFINED message could have arisen from missing the S at the end of ORDER in which case the user can add the S to the original statement immediately using the inbuilt editor.

Similarly, the statement:-

```
USES ORDERS TO DERIVE;
```

would be analysed and displayed as:-

```
USES
ORDERS                          INPUT
TO
DERIVE
;
Reject, Edit ?                  OBJECT MISSING
```

A fatal error has occurred which is reported at the point of failure. The user can only Edit or Reject the statement at this point, since Acceptance of the statement is out of the question.

4.3 Generating Reports from the MICRO-PSL Database

A sample of the reports currently available with MICRO-PSL are:-

4.3.1 Data-Process-Interaction Report (Figure 5)

This report shows the ways in which the objects in the system are either received or generated or used by the processes defined in the target system.

It is used to check on the completeness of the system definition e.g. Are there any processes which do not generate any outputs or receive any inputs?

4.3.2 Dictionary Report (Figure 6)

This report presents the definitions associated with each name used in the description of the target system. It is used by analysts to maintain the definitions of names in the database and as a tool for communication with the users of the target system.

Clearly this report serves the same function as the Data Dictionary developed using Gane & Sarson's Methodology.

continued....

4.3.3 Formatted Problem Statement (Figure 7)

This report provides a complete description in PSL of one or more names in the Analyser database. Since the FPS presents the complete information held for any name in the database, it is usually recommended that an FPS for all names be maintained as a reference and updated when changes are made to the database.

4.3.4 Structure Report (Figure 8)

This report presents the hierarchical relationships between objects in the database. It is used by analysts to maintain the consistency of any structures defined for the target system.

Using such reports individually or in combination, the analyst is able to check on the accuracy and completeness of the functional specification. Changes to the data dictionary to correct any omissions or inconsistencies are then easily accomplished on the computer.

The language used with MICRO-PSL is a subset of PSL. The subset has been chosen to cater for a teaching environment, e.g. aspects related to the management of large projects have been omitted.

Similarly the reporting facilities on Version 1 of MICRO-PSL have been restricted to the Michigan PSA reports which would be of most immediate value to trainee systems analysts. (FPS, DICT, DPI, NL, STRUCT).

Although MICRO-PSL was originally developed on an HP1000 computer, it has since been transferred to our mainframe CYBER computer to provide simultaneous on-line access to the package for a large number of users. It will be used for the first time on Systems Analysis courses by our Postgraduate students this year where we intend to use it as a Data Dictionary facility in conjunction with Gane & Sarson's DFD's.

Although we are pleased with the progress on MICRO-PSL and value the assistance it provides, it is still a long way short of the facilities which a computer could provide in the systems development process. The following section deals with the features we would like to see in an improved package.

5. Requirements for a Computer Aided Systems Analysis Teaching Package

5.1 A Consistent, Complete Teaching Methodology

Our experience of teaching of Systems Analysis during the past ten years has impressed upon us the need for a formal methodology. This should provide us with a step by step procedure to follow; which starts with the specification of systems requirements and ends with the physical systems design. Two such methodologies have recently been brought to our notice, one by Winchester (5) and the other by the ISDOS group (6).

continued....

5.2 Computer Assistance

When an acceptable methodology has been defined, it would be valuable in a teaching environment if any computer assistance could be 'methodology driven'. This would mean that a student would be required to follow a predetermined set of procedures (some or all of which might be computerised) with the computer package leading the student from stage to stage of the systems development, as each is satisfactorily completed.

In terms of individual elements of such a computer package, the advent of sophisticated graphics terminals makes it even more likely that the way forward will be through a 'picture' based methodology, rather than the current text oriented ones.

We are currently investigating the problems of developing a graphics interface to MICRO-PSL which will remove the need for students to draw the Gane & Sarson DFD's by hand.

6. Conclusion

As teachers of Systems Analysis and Design, we feel that a satisfactory teaching methodology for the systems design process has yet to be defined. We would like to see such a formalism defined as a completely separate exercise from possible computer assistance to allow us to teach the principles involved, before applying them to practical applications. Our particular interest would then be in the development of computer aids to support the proposed methodology with emphasis being placed on providing a useful set of teaching tools.

Bibliography

1. Structured Systems Analysis - C.Gane & T.Sarson, Prentice-Hall 1979.

2. The Practical Guide to Structured Systems Design, M.Page-Jones, Yourdon Press 1980.

3. Computerised Documentation in the Teaching of Systems Analysis and Design. R.J.Thomas, Computer Bulletin, June 1979.

4. ISDOS Project, University of Michigan, Ann Arbor, USA.

5. Requirement Definition and its Interface to the SARA Design Methodology for Computer-based Systems. J.W.Winchester, J.R.Hughes Aircraft Corporation 1980.

6. The use of PSL/PSA with Structured System Development Methodologies. ISDOS Project, August 1982.

February 1983

A Hierarchical System Model for Vertical Migration*

by

Gabor David
Computer and Automation Institute Budapest
Hungary

and

Wolfgang Graetsch
University of Dortmund
West Germany

Abstract

In the following paper a new system model for vertical migration purposes is presented. The model is based upon an architecture description language. It allows especially the modelling of hierarchical structures which are not only oriented to traditional software/firmware/hardware borders.

Keywords

vertical migration, system model, hierarchical structures, migration of functions and data structures

* This work has been partially supported by Deutsche Forschungsgemeinschaft (DFG) under contract Ri 367/1 and by the Hungarian Academy of Sciences.

1 Introduction

Vertical migration is a well known technique to improve the performance of a computing system. In its original form functions are moved from software to firmware. Generally vertical migration is applied to existing systems. The rearrangement of functions leads to a partial redesign of a system which may affect large parts of it. Thus system design and models for this purpose play an important role for vertical migration, too. Another goal of vertical migration is an improvement of the system structure.

There are two approaches for migration either instruction-sequence oriented or function oriented. The first one is especially tailored to the architecture interface of a computer (machine language). For the more general function oriented approach Stockenberg [Sto 78] developed a hierarchical system model which is oriented to the multi-level interpreter hierarchy (software/firmware/hardware) of a von Neumann computing system. Up to now this model is the only one for function oriented migration. For each level mapping and execution actions are distinguished. Mapping actions map flow of control and data parameters from the caller to the level of the called function and back. Execution actions refer to those steps that perform the semantic operations for the invoked function.

If we look at the design of a classical von Neumann computer there is a strong separation into control and data. We can find examples for this principle either in the CPU, separated in control unit and arithmetic unit, or even in application programs which are separated into data parts (declarations) and functions.

Function oriented migration and underlying system models only deal with the flow of control in a computer. As new VLSI technology offers possibilities for high speed memories accessible only by microprograms data migration leads to performance improvement as well as structural improvements. One major problem in VLSI technology is the data access through inter-chip connections. Thus data and functions should not be separated on different storage types but migrated together.

In this paper we first state requirements for a system model which should be a base for vertical migration. Then we discuss why the model of Stockenberg and even the improved one by Stankovic [Sta 81] is insufficient to fulfill the requirements. Then we proceed with our system model for migration.

In the fourth chapter the system model is applied to parts of the UNIX operating system [Rit 74]. Finally we discuss the quality of the model compared to the requirements.

2 Requirements

In general there is a need to have a vertical migration oriented system model for the following purposes:

- better understanding of the system architecture which is defined by functions, data, and their structure on different levels (software/firmware/hardware). Note that this definition covers disciplines like computer architecture or operating system architecture.

- identify the functions and their relations to the whole system.

- a system model should supply tools for vertical migration in the general sense where the system is multi-layered in its firmware and even hardware structure

The requirements derived from these purposes stated above are:

A system model should

1. provide a multi-level system description which is not only oriented to traditional software/firmware/hardware borders. For instance large software systems may be hierarchically structured with several levels, too,

2. offer level-independent language and language primitives, the same formal language should be used for description (specification) of any system level,

3. be realization independent,

4. support a method to identify and isolate the functions to be migrated taking control flow and data accesses into account,

5. support verification methods in order to verify functions which had been migrated and to verify that part of the system in which migration had been performed,

6. be computer aided because of the growing complexity of systems,

7. contain interfaces to monitoring tools. Performance monitors are used in order to get information about the dynamic behaviour of a system. Then migration candidates are selected.

In the following subsection we discuss why the Stockenberg/Stankovic approach only partly fulfills the requirements. The approach serves as a global system model. A detailed analysis of UNIX [Blo 82a] however showed that we can find many smaller levels inbetween not covered by their model. Thus (1) is partially fulfilled. Further their approach does not fit the criterias of level independent language primitives (2) and partly only (4). Concerning the fourth point Stankovic [Sta 81] improved their original model treating functions and data structures together. Components (modules) in his system model are interconnected according to coupling and cohesion parameters of the Structured Design methodology [Mye 76]. This heuristic methodology had been originally developed in software engineering research but poorly supports the vertical migration process of functions and data.

Requirement (3) is fulfilled as well as the the last one (7). This means that an interactive evaluation system [Sto 75] had been developed which graphically represents the system structure in connection with monitoring results. So computer aided modelling (6) is partly achieved. Finally it should be noted that no verification methods (5) are provided.

3 A New System Model based on Architecture Language

3.1 Definitions

Modern programming languages offer a module concept which can be classified as a software counterpart of a frame, the basic entity in Architecture Language (AL).

The basic ideas of AL are

- the arcitecture (including data and functions in every system level) can be described componentwise independent of the realization
- every component can be represented as a frame
- a frame can be manipulated independently from others (change, test, verify)

In our terminology a model of a system is a triplet (PDS,PF,SF)

- PDS : primary data structures used by the primary functions
- PF : set of primary functions
- SF : set of system functions

This definition is very general and covers the interfaces which enclose the analysed system. By vertical migration the set of primary functions will not be affected, migration is performed only inside.

A model of a level of a system is also a triplet (DS,EF,DF).

- DS : is the set of data structures involved in this system level
- EF : is the set of elementary functions provided by lower levels. At least EF is contained in or equal with the lowest level (PF)
- DF : is the set of defined functions for higher levels
  On the highest level DF contains or equals with SF

As an example we consider a model for the migration of operating system functions to the firmware level. In this case

PF : are functions defining the microarchitecture (for example ALU functions or functions of the memory management unit). They are used by microprogrammers and provided by the hardware

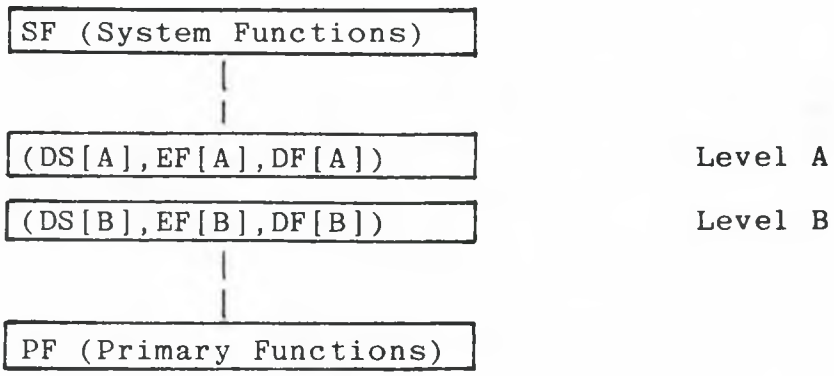
SF : system calls defining the operating system interface. These functions are either used by system and application programmers

PDS: provided by the hardware like internal and general purpose registers, main memory, processor status word.

A detailed analysis of UNIX results in a 25 level hierarchical structure [Blo 82a]. For instance an I/O system provides a set of access functions for the attached I/O devices as defined functions. Elementary functions which are needed are process synchronization, timing functions, and buffer manipulation routines. Higher levels like parts of the file system make use of the functions DF defined by the I/O system.

Let us assume two level models which are illustrated in fig. 3.1:

System:



We investigate three kinds of relations, either between

- functions (see section 3.2)

- data structures
  (DS[A] # DS[B], relations # will be discussed and explained in section 3.3)

- access relations of functions to data structures
  For data migration it is important to classify data accesses. For instance on a machine language level data accesses is performed via several addressing modes. Concerning complete functions data accesses to complex structures consists of search or update operations. More details concerning this relation can be found in [Blo 82b].

3.2 Function Structure Relationships

There are following relations between elementary and defined functions:

EF[A] => DF[A] Within a level A another set of defined functions can be derived. In other words defined functions use (call, invoke, activate) elementary functions provided for this level.

EF[A] <=> DF[B] Elementary functions of model A equal with the defined functions of model B. This relation expresses the connection between consecutive levels.

The steps of system modelling for vertical migration include:

- find pairs of consecutive level models A,B, for which elementary functions EF[A] <=> DF[B] (defined functions)
- perform inside the transformation EF[A] => DF[A]
- repeat these steps such a way that the primary functions PF <=> EF[L] equals the elementary functions of the lowest level L and the defined functions DF[H] <=> SF for the highest level H.

3.3 Data Structure Relationships

Vertical migration experiments of functions from software to firmware demonstrated that the achieved performance improvement was less than originally intended [Olb 82]. One reason is due to the fact that data which is accessed by microprogrammed functions resides in main memory. Thus read/write cycles during microprogram processing increase the execution time.

Furthermore if virtual machines should be implemented by means of vertical migration structural imbalances remains. Functions and data are separated on different types of storage (writable control store, main memory).

Our modelling approach provides a base not only for migration of functions but for migration of data structures, too. Thus the important role of data structures for system design and complex relationships between various types of data is investigated.

In the following section we will defines some data structure relations. Elementary data structures in Architecture Language notation are described in [Dav 81]. They consist of data types bit, byte, word, integer, real, character, and boolean. Structured data types can be constructed by array or record declarations. We will refer to the following data type declaration in the next sections:

```
structure rec <S[1]:T[1],S[2]:T[2],...,S[n]:T[n]>;
```

S[i]'s are selectors of record components and T[i]'s are basic data types or have been already defined. The declaration of an actual instance (REC) of this type has the form

```
structure(rec) REC;
```

Modern software development should be based upon a module concept. This means that functions and data are grouped into modules. Data can only be accessed by functions in that particular module. Modules can be grouped in hierarchical structures. They are interconnected only by function calls. Concerning this ideal structured system relations DS[A] # DS[B] defined below

would be empty. This ideal scheme is not always possible to implement because hardware related tasks (resource management) of operating systems leads to relations defined below. There are following possible relations and operations on data structures crossing level boundaries:

REFINEMENT

Formally the declaration

structure rec <S[1]:<s[11]:T[11],..,s[1m]:T[1m]>,...,S[n]:T[n]>

is a refinement of the data type "rec" in its component of type T[1] selected by S[1].

Another way is to redefine T[1] by

structure T[1] <s[11]:T[11],...,s[1m]:T[1m]>

in the same frame but in this case the data type "rec" would be refined by those T[i] (i=1,...,n) components, for which T[1] = T[i]. So these two ways to refine a data type are not equivalent.

Refinement plays an important role during system design. In the UNIX file system file names are specified by the user by path names, each subcomponent specifies a directory. The task of the logical file system is the conversion of a path name into a unique file identifier.

Within the next lower level, the basic file system, files are only identified by a unique number. Files are organized on a disk in units of 512 byte data blocks. The conversion of path names is supported by directory files containing one entry for each file. The first part of an entry is a name field (part of the path name), the second part is a unique file identifier either to specify the file itself if the path name is completely scanned or the next directory. So we can see that a directory file is a refinement of a normal file data block.

ALLOCATION

If a software function is migrated to the firmware implementation level address calculation for arrays or records have to be implemented explicitly. Normally this is done by a sequence of compiler generated machine instructions.

Additionally if we consider data migration to a new high speed memory which is only accessible by microprograms completely different storage access functions have to be used.

Let us assume that x is an already defined data type and REC is declared as above. Then

REC <Sj:x>

is an allocation statement. So REC[Sj] can be used on the current level (and on higher levels) as an x-type and on lower levels as the original Tj-type.

A realistic example in the UNIX system is the "user" data structure describing the state of a process in the system (open files, user identification, I/O operation parameters). Let us assume VM (virtual memory) as the data structure for main memory which is addressed by virtual addresses. The actual "user" structure which represents a running process is always located to the same virtual address 0140000 (octal value). This relation can be expressed as

VM <0140000 : user>

On a lower level however VM [0140000] specifies a single byte. Internally a change from one active process to another ready one can be performed changing the content of a dedicated memory management unit register. This hardware unit performs the mapping from physical to virtual addresses.

EXPANSION, REDUCTION

Again we try to motivate this relation by an example. The UNIX operating system kernel contains large data structures (records). An analysis [Hen 81] showed that many data structures are overloaded: Too many functions access the same data structures. No module concept is followed in the UNIX system. Similar experiences have been reported for large IBM operating systems. Vertical migration investigations [Blo 82b] also demonstrated this problem: If a data structure is migrated all the accessing functions have to be migrated too. But as a fast local store for microprograms is limited in size this is not possible in any case.

Originally these data structures had been designed step by step according to the hierarchy imposed by the calling relationship of functions. At a low level the structure name and a few components are fixed. Going to higher levels more components are needed which results in the final data structure design. If we want to migrate functions and data we have to investigate the original design in order to isolate data and function parts of the system.

Formally a sequence

```
structure r             ;
structure r<S[1]:T[1]>;
structure r<S[2]:T[2]>;
    ...
structure r<S[n]:T[n]>;
```

declares the same type as "rec". The single declarations in each line can be included at different levels of the hierarchy.

In the UNIX file system files can be grouped into directories. Even directories can be comprised in new ones. This results in a tree like structure of all directories and files in the system. In this scheme files and directories can be uniquely specified by path names.

Formally

```
structure filesystem <P[1]:directory,...,P[n]:directory,
                      P[n+1]:file,...,P[m]:file>
```

P[i] are appropriate path names. Data types directory and file may be already defined. For instance this file system can be expanded by

```
structure filesystem <P[k]:directory>
```

for a new path name P[k].

We say that B is a REDUCTION of A if A is an EXPANSION of B.

EXTENSION,RESTRICTION,EQUIVALENCE

These relations are meant as for sets. DS[A] is an extension of DS[B], if DS[A] contains DS[B]. In this case DS[B] is a restriction of DS[A]. If both conditions are valid they are equivalent sets of data structures.

Especially for protection in operating systems a reader can imagine the importance of relations like RESTRICTION and REDUCTION.

3.4 Frames Put Together to Make Models

A frame in AL consists of an interface, specification, and implementation part. In the interface part the data structures DS are declared. In the specification part the set DF (defined functions) is specified. The set of elementary functions is implicitly declared by the undefined function symbols of the actual frame. Furthermore there are elementary functions of AL itself (e.g. assignments, control structures) from which AL assumes a default interpretation. It is also possible to redefine them during system design.

It should be noted that the specification part will never be executed. The implementation part describes how the functions are performed.

The typical steps in AL are to put frames together

- merge two frames F[X] and F[Y] into one F[Z] = F[X] + F[Y]
  Thereby the number of defined functions but also the number of elementary functions is increased.

- invoke a frame F[Y] from the implementation part of some other frame F[X]
  Thereby for the combined frame the number of elementary functions is reduced

A model M[X] = (DS[X],EF[X],DF[X]) of a system level X can be described by means of a frame F[X].

```
FRAME F[X] (input parameters; output parameters)
INTERFACE
    description of DS[X];
SPECIFICATION
    definition of DF[X];
    description of processes activating these functions;
    activation by guarded commands [Dij 75]
IMPLEMENTATION
    may be either software, firmware, or hardware
    invocation of other frames;
ENDFRAME
```

On this base vertical migration can be performed in two different approaches:

- do not change interface and specification, change only the implementation part (for instance from software to firmware)

- restructure the system at least partly thereby changing interface, specification, and implementation parts of some frames.

Less attention has been paid in the literature to an activity which consists of deciding on the hardware/software implementation of functionality in any particular level. Once the functional design of a system is complete (interface and specification) the implementation activities should take place separately but iteratively until the desired system attributes seem to be best attained.

Chapter 4 contains an example of AL for a small part of the UNIX system. Especially the invocation scheme between frames has to be carefully investigated for vertical migration purposes:

- todays computer architectures do not utilize a firmware to software subroutine call.
  Generally in a multi-level interpreter hierarchy control flow is limited such a way that functions can only activate other ones on lower levels but not in the opposite direction. This is the reason why the hierarchical calling rule have always to be kept if functions move downwards by vertical migration.

- if there are loops in the procedure calling graph on the software level however we have to think about a redesign or a combination of all the functions involved in this loop into a single level, described by a frame. Then we can decide for an appropriate implementation, in this case either software or firmware.

- in hardware there are no software concepts like a subroutine call with a shared usage of functions. If the hardware level is involved in vertical migration the structure of the system has to be restricted in this respect

- firmware monitoring [Hol 82,Gra 82] had been especially tailored to the measurement of software functions as a base for vertical migration decisions (and requirements for architecture support).

4 Case Study of an Operating System Memory Management Function

The following small example has been taken from the UNIX operating system running on a PDP-11 computer. Implementation details have been omitted as far as possible in order to be understandable for a reader not familiar with details. The function "memory-allocate" performs an algorithm for main memory management according to a first fit strategy. Although the example is very small it should be general enough. Furthermore it is a well known algorithm known from operating systems. Two major problems are discussed along with the presentation of this simple example

- function refinement problem
  Up to what extend should a function be refined (decomposed) into smaller ones before parts of it are migrated to a lower level (which functional refinement is appropriate for a certain implementation level).

- the distinction between strategies and mechanisms
  A general accepted rule for migration states that only so called mechanisms should be migrated [Bro 76]. Strategy routines should be avoided for migration to lower implementation levels because changes are more likely to occur.

If we consider the resource main memory it can be modelled as

```
structure M < [1:248K] : bit(8) >
```

For the declarations a shorter notation is used (1K equals 1024). Memory as seen from the machine architecture interface is accessed usually via a memory management unit, thereby specifying virtual addresses instead of physical addresses.

```
structure VM < kernel-space : [1:56K ] : bit(8)
               user-space   : [1:192K] : bit(8)
               i/o-page     : [1:8K]   : bit(8) >
```

The selection either of kernel space and i/o page or user space depends on the actual mode of operation. One task of the operating system is the management of the user-space among a varying set of processes. The frame "memory-allocate" manipulates a data structure "user-space-map" for this purpose:

```
structure user-space-map < [0:N]:
    < size : bit(16),    /* size of a free storage block  */
    < addr : bit(16) >> /* start address of a free block */
```

Continuous storage blocks are obtained from this pool.

```
FRAME memory-allocate(map,required-size ; map,return-address)

INTERFACE
   struct(user-space-map) map;
   bit(16)                required-size,return-address;

SPECIFICATION

   bit(16) i;

   if (required-size >  0)

      then search-for-enough-space (map,required-size,i);
           allocate-resource (map,required-size,
                              return-address,i);
           compress-resource-map (map,i);

   if (required-size <= 0)

      then return-address := 0;

IMPLEMENTATION

      two-comp-table-search (map,required-size; i);
      allocate-table (map,required-size,i; return-address,i);
      compress-zeros (map,i; map);
      test-zero (required-size; return-address);

ENDFRAME
```

In the first line input and output parameters separated by a semicolon are specified and further precisely defined in the interface part. Data structure "map" is changed by the frame. So it has to appear either as input and output parameter.

In the specification part two guarded commands controls the activation of the following processes. This implies that the sequences defined subsequently can be executed in parallel. AL specification offers on each level actually modelled the possibility for parallel processing. As vertical migration is a technique for monoprocessor systems parallelisms is a possiblity only for lower levels. Firmware structures for instance offers possibilities for parallel execution.

Mainly in the specification of this frame a functional decomposition is performed. The functions specified here must be executed by the invoked frames of the implementation part which is written in a data flow language. The invoked frames with actual parameters are given in fig. 4.1.

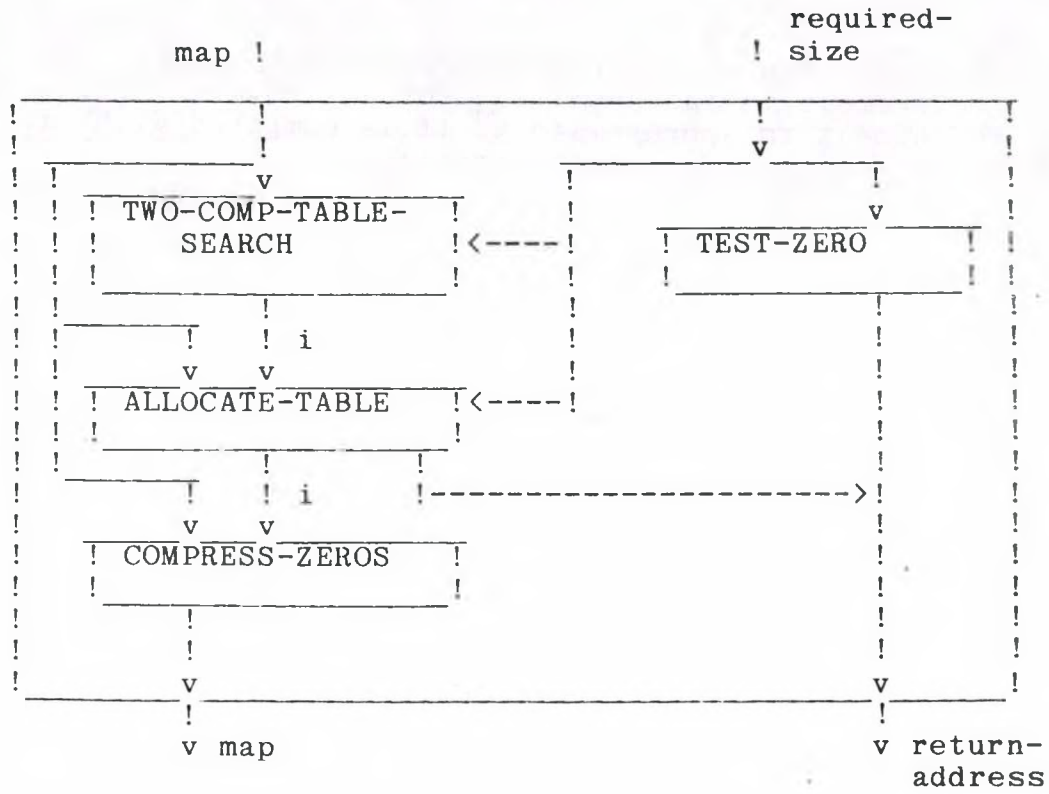


Figure 4.1: Data flow diagram of memory-allocate

In the following we will describe the interface and specification parts of the invoked frames.

```
FRAME two-comp-table-search (map,required-size; i);

INTERFACE
   struct (user-space-map)  map;
   bit(16)                  required-size,i,m;

SPECIFICATION

   function search-for-enough-space (map,required-size, i);
   begin
      m := 0;

      while(map[m][size] < required-size && map[m][size] > 0)
        m := m + 1;

      if (map[m][size] = 0 ) i := -1;
      if (map[m][size] >= required-size) i := m;
   end

   search-for-enough-space (map,required-size  ,i);
ENDFRAME
```

This frame implements a search algorithm looking for the first free slot which fits the requirements. If the strategy has to be changed for instance to best fit only this frame is affected and has to be modified.

For a software implementation the reader will easily verify that this refinement is too detailed. For a firmware version however this refinement is appropriate if it is compared with the complexity of a normal machine instruction.

```
FRAME allocate-table (map,required-size,i; return-address,i);

INTERFACE
   struct (user-space-map)  map;
   bit(16)                  required-size,i,m,return-address;

SPECIFICATION

   function allocate-resource (map,required-size,
                               return-address,i);
   begin

      if (i = -1)  return-address := 0;

      if (i > 0 )  return-address := map[i][addr];

               map[i][size] := map[i][size] - required-size;
               map[i][addr] := map[i][addr] + required-size;
   end

   allocate-resource (map,required-size,return-address,i);
ENDFRAME


FRAME compress-zeros (map,i; map);

INTERFACE
   struct (user-space-map)  map;
   bit(16)                  i,k;

SPECIFICATION

   function compress-resource-map (map,i);
   begin

      if (i > 0 && map[i][size] = 0)
      then begin
             k := i + 1;

             while (map[k][size] > 0)
             begin
                map[k-1] := map[k]; k := k + 1;
             end;
           end;
   end;

   compress-resource-map (map,i);
ENDFRAME
```

Frame "compress-zeros" shifts the memory map from the end of the table to the empty slot (if any). Thereby the end of the user space map must be indicated by a zero field. The implementation

parts are omitted in this example. In the specification parts functions are activated without any guard. In this case a shorter notation is used which means that the guard is always true.

By now functional, structural or topological, and the modeling completeness can be checked:

- Functional completeness
  In the frame memory-allocate the elementary functions are search-for-enough-space, allocate-resource, and compress-resource-map. The implementation part invokes frames, in which these functions are defined functions. The definition there use only AL-defined functions which are elementary ones. So there is not any function open in this context. The functional completeness is checked using the SPECIFICATION parts only.

- Data structure completeness
  In our example we start from the IMPLEMENTATION part of the frame memory-allocate (see fig. 4.1). For each invoked frame it is easy to verify that the parameters are matched. This type of completeness is frequently refered as the topological one.

- Modeling completeness
  From the SPECIFICATION parts of the frames two-comp-table-search, allocate-table, compress-zeros, and test-zero and from the IMPLEMENTATION part of the frame memory-allocate it can be verified that the specification of memory-allocate will be executed correctly by the implementation. The underlying assumption is that the invoked frames will do what is stated in their SPECIFICATION parts. This completeness is a local one because this is applied to the frame memory-allocate only.

5 Conclusion

We finish this paper comparing the requirements and explaining how a system model based on Architecture Languages satisfies them:

- multi-level system description (1), realization independency (3)
  Frames are connected only via implementation parts. They can invoke each other. Actual parameters are specified by data lines. Frames are the components realized either by software, firmware, or hardware. Specification parts of diffent frames are invariant against changes of the implementation part. Thus multi-level system description is achieved which is also realization independent in its interface and specification part.

- level independent language (2)
  is a requirement which is achieved for the interface and specification part. Data lines connecting frames are formally defined by data structure declarations based upon the smallest unit which is a bit. Specifications are defined in a formal language independent of the realization.

- identification and isolation of functions and data (4)
  There are no global variables in the language concepts of AL. Each data structure used by a function must appear in the interface part. Functions can invoke other functions with the specified parameters of the interface. Data structure relations have been defined precisely in section 3.3. In this form they can appear in different interface parts of frames.

- verification (5)
  In AL the following verification methods are assumed: simulation and check for the functional and data structure completeness. The check for completeness is a part of the syntactical analysis, so it can be automatically be performed. Another method is the logical completeness based on guards.

- computer aided methodology (6)
  AL is not a manual method but a computer aided one because the description of frames is based upon formal languages.

- interfaces to monitors (7)
  Dynamic parameters of monitoring are assigned to functions in the specification parts. On this base vertical migration decisions can be made.

6 Literature

[Blo 82a] Block, H., Graetsch, W., Kaestner, H.
Documentation of the UNIX System
Internal Report 4, Project Vertical Migration, University of Dortmund 1982

[Blo 82b] Block, H.
Untersuchung zur Migration von Datenstrukturen
Diplomarbeit, Universitaet Dortmund, Oktober 1982

[Bro 76] Brown, G.E., Eckhouse, J.R.
Operating System Enhancement Through Microprogramming
Sigmicro, March 1976, Vol. 7, (29-33)

[Dav 80] David, G.
Restructurability - A Tool for System Development,
Proc. IFIP Working Conf. on Firmware, Microprogramming and Restructurable Hardware, Linz, 1980 North-Holland, 1980, (135-158)

[Dav 81] David, G., Losonczy, I., Papp, S.D.
Language Support for Designing Multilevel Computer,
Systems. Handler, W. ed. CONPAR'81, Springer Verlag Lecture Notes on Computer Science, Vol 111, 1981, (85-100)

[Dij 75] Dijkstra, E.W.
Guarded Commands, Nondeterminancy and Formal Derivation of Programs
Comm. ACM, 18, 8(1975)

[Gra 82] Graetsch, W., Kaestner, H.
Firmware Monitoring of the UNIX Operating System,
Internal Report 5, Project Vertical Migration, University of Dortmund 1982

[Hen 81] Henry, S. Kafura, D.
Software Structure Metrics Based on Information Flow
IEEE Trans. on Software Engineering, Vol. SE-7, No. 5, Sept. 1981

[Hol 82] Holtkamp, B., Kaestner, H.
A Firmware Monitor to Support Vertical Migration Decisions in the UNIX Operating System,
15th Annual Workshop on Microprogramming, Oct. 1982
Palo Alto, California

[Mye 76] Myers, G.J.
Reliable Software Through Composite Design
Mason/Charter Publishers, London, 1975

[Olb 82] Olbert, A.G.
Crossing the Machine Interface
15th Annual Workshop on Microprogramming, Oct. 1982
Palo Alto, California

[Rit 74] Ritchie, D.M., Thompson, K.T.
The UNIX Time Sharing System
Comm. ACM, Vol. 17, No. 7, (365-375)

[Sta 81] Stankovic, J.A.
Improving System Structure and its Affect on Vertical Migration
Microprocessing and Microprogramming 8 (1981), 203-218
North-Holland Publishing Company

[Sto 75] Stockenberg, J.E., v. Dam, A.
STRUCT Programming Analysis System
IEEE Trans. Software Engineering, Vol. SE-1, No. 4, Dec. 1975

[Sto 78] Stockenberg, J.E. van Dam, A.
Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/Software Systems,
Computer, Vol. 11, No. 5, (39-50)

METHODOLOGY for SYSTEM DESCRIPTION USING THE SOFTWARE DESIGN & DOCUMENTATION LANGUAGE

by

HENRY KLEINE

ABSTRACT

The Software Design and Documentation Language (SDDL) can be loosely characterized as a text processor with built-in knowledge of, and methods for handling the concepts of structure and abstraction which are essential for developing software and other information intensive systems. Several aspects of system descriptions to which SDDL has been applied are presented and specific SDDL methodologies developed for these applications are discussed.

INTRODUCTION:

The Software Design and Documentation Language (SDDL) [17], originally conceived as a simple, convenient pseudo code processor for developing program designs has evolved into a more sophisticated tool which can now be applied to a broader range of software development tasks. The evolution of SDDL includes many improvements to the language and the computer processor but the primary growth is in the area of new discoveries in methodology [18,19]. Thus, SDDL capability has expanded upward in the hierarchy of system description actions to include activities such as the specification of general system requirements and program functional requirements. It has expanded downward to include documentation and pretty-printing of structured programming languages, and it has expanded laterally to include methods for describing rules and formats for specifying program input data. It has even developed tangentially to include methodology for handling genealogical family trees. In general, the scope of SDDL has grown in the direction of handling information that is best conceived and represented in a structured format.

This paper presents the results of one phase of research carried out at the Jet Propulsion Laboratory, California institute of Technology, under Contract NAS7-100, sponsored by the National Aeronautics and Space Administration.

As the author of SDDL I am pleased to acknowledge my contribution to this technology but the greater credit for its current utility as a software development tool belongs to the many users who have contributed imaginative methodology and suggestions for improvements in the capabilities of the processor itself.

SDDL OVERVIEW DESCRIPTION

SDDL can be described simplistically as a language processor with built-in knowledge of and methods for handling the concepts of structure and abstraction which are fundamental to software development specifically and to the description of information intensive systems in general. SDDL is comprised of a language, a processor, and methodology for their use. The SDDL syntax consists of a small set of keywords which are used to create design structures in the manner of Structured Programming [1,8,9,11] and a set of directives which provide the user with control of the SDDL processor formatting functions.

Since SDDL only formats the input and does not produce executable code, only two structures, the Module and the Block, are needed to specify a design. Modules are used to represent abstractions which are complete and independent enough (a subjective user opinion) to be treated as separate entities. Modules are given descriptive names and their interrelationships are stated explicitly by means of a module invocation statement reference (analogous to a programming language CALL statement) within the module. Blocks are the lower level constructs, such as iterations, conditionals and sequences which are used to represent concepts or algorithms internal to a module. Both kinds of structures require an initiator keyword statement and optionally may have a terminator, a substructure, and/or an escape keyword statement. A keyword statement is a statement which begins with a predefined keyword such as CALL, IF, ELSE, etc. The user can use pre-established sets of keywords or may define keywords specifically tailored for the task at hand.

The actions taken by the processor in response to keyword statements are quite simple but effective for communicating structured information. Indentation of statements within structures and flow lines which highlight both structure escapes and module invocations provide a visual, two-dimensional information display that is topologically equivalent to a conventional flow chart. This technique captures most of the advantages offered by flow charts without their attendant disadvantages. This formatting capability is illustrated throughout the example in Appendix A.

SDDL Directive statements provide the user with the means to control document formatting and to direct the production of cross reference tables and other document summary information. The functions performed by the SDDL processor are summarized in the following list:

Document Formatting:

Line numbering for input file editing.
Indentation to display structure logic.
Structure logic error detection.
Flow line arrows to accentuate structure escape statements.
Flow line arrows with page references to module invocations.
Special handling for title pages and commentary text segments.
Input and output line continuation.
Line Splitting (partial right justification of output lines)

Document Summary Information:

Table of contents.
Module invocation hierarchy (tier chart).
Module cross reference table.
User selected cross reference tables.
General cross reference table (includes all identifiers).

Processor Control Capabilities:

Page margins, length, numbering, heading and ejection.
Structure indentation amount.
Deletion of leading blank characters on input lines.
Input line numbering sequence.
Keyword definition.
User cross reference table definition.
Specification of a label field on the input file.
Specification of a sequence number field on the input file.
Options for suppressing selected processor features.
Selection of comment delimiter characters.

The basic formatting functions of SDDL, which are easily mastered, are all that the user needs to begin laying out the specification of a design. With SDDL directives the user can define new structures, control the capture of identifiers for cross referencing, control document formatting, and selectively suppress the generation of summary reports. These capabilities in themselves have value only as computer automated documentation conveniences, but "convenience" can have a much greater meaning when it has the effect of freeing the document creator from the many tedious, repetitious tasks not related, but unfortunately necessary for document preparation. Furthermore, when these conveniences are augmented by methodology that facilitates the conveyance and understanding of the elements of a system's description they also assist the document reader in the same way. Elimination of these tedious, repetitious tasks by means of computer automation is like oiling the interacting parts of a mechanical device to reduce friction. Just as mechanical drag expends energy for work which does not contribute to the direct purpose of the machine, so "Cerebral Drag" drains energy from both the writer and the reader for work which is not directly related to creating or understand a document. Thus, these simple capabilities, when applied with imagination and insight in the development of the documentation can be utilitarian and functional.

EXAMPLE APPLICATION OF SDDL

The system selected to illustrate the use of SDDL is the SDDL program itself. Because of obvious space limitations the example document (Appendix A) has been greatly shortened by 1) omitting some of the system description capabilities to which SDDL has been successfully applied, and 2) by carrying the level of the documentation deep enough to convey just the essence of meaning and style.

The system description areas exemplified below include:

1. Statement of the overall objective of the system
2. Hierarchical Input-Process-Output development (HIPO)
3. SDDL detailed design
4. Pascal source code
5. SDDL program invocation command
6. Formal syntax definitions

Some system description capabilities which have been omitted are:

* Function requirements specification
* Genealogical family tree
* Documentation of program source code for SIMSCRIPT II.5 and FORTRAN 77.

GENERAL COMMENTARY:

In the following discussion of the example problem some SDDL capabilities which apply in general to all usage (e.g., title pages, table of contents) are mentioned where they first occur in the document. The page and line numbers noted in the discussion refer to page and line numbers of the example document in Appendix A. Page references are given at the top of each subsection and the line references appear within the text enclosed in angle brackets, < >.

Unnumbered Title Page:

SDDL provides processor directive statements that allow the user to delimit a group of input lines which will be the content for a title page. The processor enters the title page heading into the document Table of Contents and boxes and centers the contents of the title on a new output page. The choice of the character to be used to form the box is a user option.

Page I: Table of Contents:

The Table of Contents is produced automatically by SDDL. Note that there are three levels of indentation: The top level is for titles, the second level for modules, and the bottom level for module substructures.

Page 1: Statement of the Overall Objective of the System:

The purpose of this module is to introduce the objective of the system in a general way as an explanation for the new reader. It can also serve a stronger role as an informal contract between customer and implementer stating the agreement regarding the overall purpose of the effort.

The text of the statement of objectives is automatically boxed as was the title. Text boxed in with asterisks (referred to as a TEXT BOX) within a module as shown (29-42) differs from a TITLE BOX in that it is printed in the exact context in which it was found and is not centered on the page or entered into the Table of Contents.

Note that on this and on all other output pages, the line numbers printed on the left margin correspond exactly to the line numbers of the input source file. This precise correspondence greatly facilitates the source file editting process.

HIERARCHICAL-INPUT-PROCESS-OUTPUT DEVELOPMENT

Pages 2 - 4: Hierarchical Input-Process-Output (HIPO) [33]:

Throughout this document, heading lines unique to each section are printed at the top of all pages. This was accomplished by inserting a SDDL heading directive at the beginning of each section specifying the heading text desired.

Page 2: Title Page Introduction to the HIPO Section.
(Omitted to conserve space)

Page 3: Top Level of the HIPO Development:

A TEXT BOX is used to provide a general description of the module content. The contents of this module are structured to represent the three parts of the HIPO concept. INPUT, PROCESS, and OUTPUT have been previously defined as SDDL keywords so that the processor can automatically provide indentation to display the HIPO structure. Input and output data elements are numbered sequentially and the numbers are enclosed in square brackets for automatic entry into the "HIPO DATA SETS" cross reference table. The square brackets and a corresponding title for the cross reference table have been previously defined by a SDDL directive statement. The word EXECUTE has also been defined previously as a module invocation keyword. This causes the processor to add the right hand arrow to the statement and enter the

page number of the module that was referenced (68 & 69). The short right hand arrows (67,70) are not produced automatically but are part of the source input line. Their purpose is to point to the output data element which is produced by the stated action. To increase the visibility of the data element number it is automatically justified at the right hand margin. The right justification function was triggered by a user defined special character placed in the input statement.

Page 4 Top half: HIPO Description of the SDDL First Pass

This module is called from the HIPO_DEVELOPMENT module (68). It is developed in the same manner as above to describe the data processing steps of the first pass operation.

Page 4 Bottom half: HIPO Description of the SDDL Second Pass

This module is also referenced by the top level module (69). It describes the actions of the second pass operation.

Since this module and the preceding one are small they were both printed on a single page to conserve space. This was done with the SDDL page compression directive. This directive does not appear in the document but the reader may note that the input line number of this directive (101) does not appear in the printout.

Page 21: Module Invocation Tree, Summary Report

The Module invocation tree (tier chart) provides a summary report which displays the module invocation hierarchy. The HIPO elements (2-4) are segregated from the other elements because they are not linked by module invocation statements. In this example the tree is quite shallow because the lower levels of the example have been omitted.

Page 23: Module Cross Reference Table, Summary Report

The modules used for the HIPO description also appear in the alphabetically ordered Module Cross Reference Table. Note that the HIPO modules have all been prefixed with "HIPO_" so that they all appear together in the alphabetical listing.

Page 25: Cross Reference Listing for HIPO DATA SETS

The data elements of the HIPO description which were prefixed by a number enclosed in square brackets are all included in this cross reference table. An SDDL directive was used to designate the underscore and both square brackets to be used to form identifiers. This was done as a matter of style.

SDDL DETAILED DESIGN

Pages 5 - 9: Pseudo Code Development of the SDDL Detailed Design

Page 5: Title Page Introduction to the Detailed Design Section. (Omitted to conserve space)

Page 6: Description of the Data Structures of the Design

As in the previous modules a TEXT BOX (131-135) is used to provide a brief description of the purpose of the module. Such descriptive commentary, now widely recognized as good programming practice, is well set off by the enclosing TEXT BOX.

The data structure for the SDDL processor includes hierarchies of scalars, arrays, and linked lists of records. The hierarchical nature of the data is captured by means of indentation. To produce the indentation automatically, an SDDL directive statement was used to define keywords LIST and MEMBER as block initiators. For reasons of style, a special non-printing directive statement (155) was used to close the block structures rather than use a block terminator keyword. Thus, the indentation for the structure (143-154) specifies a linked list comprised of entities, dx.ENTRY (145), each of which individually "owns" another linked list comprised of entities, dx.REFERENCE (152).

The data element names are all prefixed with a two character notation to identify each datum with the specific area to which it belongs. For a complex design document this prefix is very helpful because it makes the context and relationship of the datum immediately clear. Although this prefix notation may initially be bothersome to the reader it quickly becomes natural to ignore it while reading the document and yet the important information is always present when, as is often the case, it is needed.

Data definitions further augment the document and, where the information is available, data types, ranges, default values, etc. can be added.

Page 7: Top Level of the Detailed Design Development:

This module describes the top level of the design as a number of steps to be performed in sequence. To emphasize the sequential nature of the design a block structure comprised of initiator keyword FIRST and substructure keyword NEXT has been defined. This construct is defined as a "Comment Structure". The individual FIRST-NEXT lines are obviously comments and the entire construct is a structure in the sense that the statements following each comment pertain to and are within the scope of the preceding comment. The reader may note that a clear overview of a module is easily gained by examining the TEXT BOX and the sequence of FIRST - NEXT statements. Furthermore, by including these same constructs as comments in the final program code

a well documented link is established between the design and the code documents. Examples of this method can be found on line pairs <173 : 293>, <178 : 296> and <229 : 391>.

This module provides another illustration of the module invocation statement <179,188>, in this case using CALL as the invocation keyword. It also illustrates the SDDL actions taken when an output line exceeds the defined page width <188>. Note that the continued part of the line does not have a line number since it is part of the preceding input line <188>.

Page 8: Detailed Design for the First Pass Program

This module illustrates the action the SDDL processor takes when modules referenced in invocation statements do not exist in the document <201,203,206, 213>. The blank appearing in the page number field give a clear reminder that the module was omitted. This module also illustrates the use of conditional and iteration constructs. The ELSE keyword <208>, has been defined as a substructure of the IF block construct. This causes the processor to un-indent to the level of the corresponding initiator statement <205>, print the line, and then re-indent to continue the structure. The keyword ENDIF is used to terminate this structure <214>. The same structure <209 - 211> illustrates structure nesting.

Page 9 Top half: Design for the Second Pass Program

This module references data elements defined in the HIPO section <230,231,233,239,242>. These references have also been automatically captured in the cross reference tables.

Page 9 Bottom half: Detailed Design of the Statement Input Module

This module illustrates the effect of the block escape statement. The keyword EXITLOOP <267> has been defined as an escape for the LOOP - REPEAT block construct. This causes the processor to produce a left arrow to the level of the parent construct calling attention to the escape from the construct.

SDDL PASCAL CODE

Pages 10 - 14: Pascal Source Code Processed Through SDDL

SDDL is well suited for processing valid program code written in a Structured Programming Language [21,31] since the user, by means of the keyword definition directives, can easily define structures to match the syntax of the target language. This keyword definition step usually requires no more than ten directive statements and in the case of Pascal the structures are predefined. The SDDL processor itself was originally written in the SIMSCRIPT II.5 Programming Language [15] and later improved and written in Pascal [12,26,29,32].

The code modules shown below are excerpts from the Pascal version. The Pascal language did present some problems that required the creation of a few new SDDL techniques and some Pascal coding style conventions. The primary change was due to the Pascal block structure which permits procedure nesting. Since this is not possible with SDDL it was necessary to use the Pascal forward reference capability to predefine the procedures in order to avoid having to nest them. One could argue that the resulting document is easier to read and understand because the declarations and the programming in an outer block are not separated by the insertion of nested inner blocks. Yet the relationships between the modules is clearly shown by means of the module invocation references generated by SDDL. The use of the forward references also allowed the code modules to be presented in the document in a top-down order that otherwise would not have been possible.

The data declaration section of the program has been omitted from the example below since its representation in SDDL did not use any special features.

Page 10: Title Page Introduction to the Pascal Code Section
(Omitted to conserve space)

Page 11: Pascal Code for the Top Level Driver

As shown in the previous sections, each module begins with a brief description enclosed in a TEXT BOX. In this case, since the source lines are valid Pascal code it was necessary to enclose the entire box in Pascal comment delimiters ‹285,291›. Note also, that since Pascal makes no provision for the declaration of a Main Program, a comment statement has been added for this purpose ‹284› and for a terminator statement ‹313›.

Structures BEGIN-END ‹292,312›, and IF-ELSE ‹297,299› used in this module are automatically predefined and available to the user. Note that the IF construct requires a terminator, (ENDIF), which has been entered with a Pascal comment ‹311›.

Pascal and SDDL syntax differ in that the latter requires a specific keyword to call a module and Pascal does not. Thus the keyword, (CALL), which is a Pascal comment, has been used for this purpose ‹301,303,304,305,308,309›.

The (FIRST - (NEXT comment structure discussed earlier is included here to maintain the correspondence between detailed pseudo-code design and implementation code.

This procedure also demonstrates the use of revision notation ‹297,300›. The processor automatically right justified the revision notes against the right margin and captured them for entry into the cross reference table on page 27 under the title "REVISIONS".

Page 12: Pascal Code for the First Pass Operation

This procedure demonstrates the capability to define specific indentation for certain structures. In this case, to avoid excessive indentation, the PROCEDURE structure was defined with zero indentation. Furthermore, there is no terminator keyword defined for the PROCEDURE structure. This was done because Pascal uses the word END for many terminators and it was elected to use END to match with BEGIN since this structure receives the greatest use and benefits most from having a terminator keyword. The BEGIN-END construct was used here to form an iteration structure <325,326,339>.

Page 13: Pascal Code for the Statement Reader Module

This procedure demonstrates the use of program variable names that match corresponding names in the design document <370>.

Page 14: Pascal Code for the Second Pass Operation

As on the previous page, most of the body of this procedure has been omitted to conserve space. An example of how the Pascal scope statement is structured using the BEGIN-END construct is shown <392,393,401>.

SDDL PROGRAM INVOCATION COMMAND

This section provides a description of the sequence of input that the user must provide to command the execution of the SDDL processor. As is often the case, the explanation of the input to the command is much more complex than the command itself. This is especially so when the input instructions are very short, as in this case, where only the execution command word and the names of the input and the output files are required. Nontheless, the full explanation is necessary in order to cover non-obvious aspects of the command such as allowable inputs, defaults, and error situations.

Although the instructions presented in this example are few and simple, they perform the same function as would any large, complicated user's guide which describes the input sequence and allowable data values for the specific input data required to run the program. Another example of an input specification document can be found in the Formal SDDL Syntax Definitions in the next section. There the specifications are different only in that they are given as general rules rather than detailed instructions. It can be seen that in each of these cases the input specification document is a data preparation "program" which must be executed by the user rather than a computer. This program differs from other computer programs only in its programming language, natural English, and the processor, a human being. Since the design and structure of this program can be developed with the same concepts as a computer program it follows that SDDL can be used advantageously in this task as well. It is this rationale that led to the use of SDDL to develop the "Input Format

Specification Document" methodology which, in a greatly simplified version, is discussed below.

Page 15: Title Page Introduction to the SDDL Execution Command
(Omitted to conserve space)

Page 16 - 17: Input Procedure for the SDDL Execution Command

Since it is necessary to define the input procedure as a sequence of operations, the FIRST-NEXT comment structure <422,425,432,450,457,472> was used to create an outline of the steps. The first step directs the user to enter the processor invocation command. The word ENTER <423,426,433,451,458> was used to tell the user what must be typed. To emphasize the fact that a user input is required an SDDL directive was used to define ENTER as a module escape keyword. This causes the processor to produce the arrow to the left hand margin as shown. Notice that this use of the escape statement is different than previously where it was used to indicate an escape from a functional control construct. Here it is used simply to call the user's attention to the fact that input is required at this point. In another important way, however, both examples are the same in that their purpose is to call the reader's attention to an important bit of information. Since the sole function of documentation is to provide information, it is wise to use every technique available to enhance document readability and reduce cerebral drag.

Because of the nature of the input, the next step <425-430> happens to be a user option. The information for this step belongs at the same level as the rest of the FIRST-NEXT structure so the keyword OPTIONAL <425,450> was added to the structure definition as a synonym to NEXT to maintain this consistency.

In the event that the user exercises this option the processor must provide a response as indicated <429>. Note that for emphasis the computer's response was enclosed in a TEXT BOX but in this case an angle bracket, >, was selected for the boxing character. This was easily accomplished with the boxing character option available with the SDDL Text directive.

The next step, which calls for the entry of the name of the user's input file <432,433>, presents some complications since certain defaults are permitted and error conditions could occur as a result of faulty input. These exceptions are explained by NOTE statements <435-448>. To call attention to this important information the word NOTE was added to the FIRST-NEXT structure definition as another synonym to the NEXT keyword. This causes the processor to bring the statement out to the same level of indentation as the other elements of this structure. The selection of this or another technique to emphasize a line is strictly a matter of user's imagination and style.

With the instructions for entering the name of the user's input file completed, the next input required is the name of the output file. Here the user has the option <450-455> of starting a new input line before entering the output file name.

The next step directs the user to supply a file name <457-458> or use the defaults supplied by the command and explains the default mechanism employed <459-470>.

Finally, the last step describes the actions taken to process the user's document and signal the completion of the processing step <472-483>.

FORMAL SYNTAX DEFINITION

Pages 18-20: Top Down Definition of the SDDL Syntax

This final section of the example document presents the top levels of a top-down structured formal definition of the SDDL syntax. Formal language descriptions are highly structured documents and it is therefore not the least surprising to find that SDDL can be used effectively to describe its own syntax. The document here, as in the previous section, makes important use of the FIRST - NEXT construct including the added keyword, OPTIONAL, to specify the order in which the parts of the language must be used. Another construct added is the SELECTION - OR - END_SELECTION_OPTIONS for situations where the user is required to select one item from a list of alternatives. These and other techniques are discussed below in the context of the example.

Page 18: Title Page Introduction to the Syntax Definition
(Omitted to conserve space)

Page 19 Top half: Top Level Syntax Definition

As noted previously, the FIRST - NEXT construct <511,514> gives the reader a quick preview of the contents of a module. In this case it shows that at the top level there are only two steps. The first is a special case of an optional output suppression directive which, if used, must be the first line of input. The word SEE was defined as a module invocation keyword to use for referencing this and other directive definitions <512>. The second step of this module defines the entire syntax in three high level parts. The ITERATION - END_ITERATION <515,523> structure specifies that they may be used as often as desired, and the SELECTION - OR - END_SELECTION_OPTIONS structure <516,518,520,522> shows that they may be used in any order. The three lower level parts, which are fully defined on later pages, are pointed to by module invocation statements <517,519,521> which use an asterisk, *, for the invocation keyword. Here again, the choice of the asterisk for this purpose is strictly a matter of user preference. This section of the document also uses the underscore, period, and dash characters as concatenation marks to form single identifiers out of two or more words <509,512,517>.

Page 19 Bottom half: Syntax Definition for a Title Page

This module, which was referenced from the preceding module <517>, defines the syntax for specifying a title page. Here again the FIRST - NEXT structure, with a slight variation for a special effect, provides a crisp overview of the three parts of the title page construct. The variation was the definition of keywords *FIRST - *NEXT (note the asterisk as part of the word) as substructures for the module initiator keyword instead of an independent structure. This was done so that the sections named in this way <530,543,548> would also appear in the Table of Contents. The FIRST - NEXT comment structure is used in the usual way <531-541> to define the subparts of the directive statement. In this case a terminator keyword, END_STATEMENT, was added to the FIRST - NEXT structure to terminate the definition of the title initiator directive statement <541>.

Page 20: Syntax Definition for a Module

The specification of a module construct is similar to a title page. As can be seen from the high level *FIRST - *NEXT statements, it is comprised of three parts (initiator<559> - body<576> - terminator<595>) but the subparts in this case are more complex. The first part of the initiator statement definition requires an initiator keyword <560> but since the choice of the keyword is a user option, no specific word can be shown in this context. Therefore, the syntax description document supplies a reference to a submodule which provides a display of the available built-in keywords and structures. The next part of the module initiator statement shows that an optional "noise word" to enhance the appearance of the statement, may be entered next on the line. Of the three available choices, FOR<564>, TO<566>, and any punctuation character<568>, the first two are obvious but the third requires additional clarification so a reference is made to a lower level module for a full explanation.

The first and third parts of the module structure definition consist of definitions of single SDDL statements. The middle part, which is the body <576 - 593> of the module construct, gives the reader a list of the essential elements of SDDL and provides page references to lower level modules where complete detailed definition of the syntax can be found.

SDDL SUMMARY TABLES

Page 21: Module Invocation Tree (Tier Chart)

The Module Invocation Tree, or tier chart, uses indentation to show the "caller --> called" relationships among the modules. The tree is formed by listing all called modules under its caller at the next lower level of indentation. Line numbers shown at the left of the page are used for referencing back to previously completed branches.

Page 22: Module Invocation Tree Continued (Omitted)

Page 23: Module - Cross Reference Listing

This summary is an alphabetical, "called --> caller" report of all of the modules identified in the document by either a module initiator or a module invocation statement. For each entry in the table a list of line numbers, module names and page numbers of the modules where the entry appeared is given.

Page 24: Module - Cross Reference Listing Continued (Omitted)

Page 25: Data Items - Cross Reference Listing

This table shows how the prefix used to identify a particular class of data causes all the elements of that class to be grouped together because of the alphabetical ordering.

Page 26: HIPO Data Elements - Cross Reference Listing

This cross reference table was set up to capture the data elements of the SDDL HIPO development. The punctuation characters defined to be used to form identifiers for this table are the left and right square brackets. The underscore was defined as a concatenation character which does not produce an entry into the cross reference table. This, and the other cross reference tables are structured exactly as the module cross reference table discussed above.

Page 27: Revisions - Cross Reference Listing

The last table contains only two entries corresponding to program revisions. Some examples of other uses for cross reference tables like these are to capture references to notes for program rehosting, technical memoranda, technical liens, and requirements items.

CONCLUSION

Throughout the development of a system description, the SDDL design document should always represent the definitive word on the current status of the ongoing, dynamic development process. It is essential that this document be easily updated and readily accessible in a familiar, informative, readable form to all members of the development team. This design document is the medium of communication between designer's creative thinking and the receiver of this information. In creating such a document there is a trade-off between applying minimal effort, which increases the reader's burden, and applying a great amount of effort which minimizes the reader's task. For any serious task the efficient choice is to minimize the reader's effort since this is the task repeated most often. Even the writer must also read the document many times over. By automating many of the tedious repetitious chores which get in the way of productive effort, SDDL helps the writer produce a document that is structured and formatted in a way that also reduces the effort required by the reader. Thus,

the purpose of SDDL is to provide a bridge between the software developer and the reader which will reduce effort and enhance effectiveness for both.

The importance of readable documentation cannot be overemphasized. Reducing the cerebral drag in the reading of a complex, information intensive document greatly enhances its effectiveness as an instrument for reconciling misunderstandings and disagreements in the evolutionary development of all aspects of the system description. The structure formats, page references, and cross reference tables produced by SDDL make the structured walk-through technique [30] for joint verification of the design concepts a practical reality. The design document also supports project management by providing current documentation of progress and recording task responsibilities.

REFERENCES and BIBLIOGRAPHY

1. Baker, F.T., "Structured Programming in a Production Programming Environment," IEEE Trans. on Software Engr., Vol. SE-1, No. 2, pp. 241-252. June 1975.

2. -----, "Chief Programmer Teams: Principles and Procedures," IBM Report FSC71-5108, Fed. Sys. Div., Gaithersburg, Md., June 1971.

3. -----, and Mills, H.D., "Chief Programmer Teams," Datamation, Vol. 19, No. 12, pp. 58-61, Dec. 1973

4. Brooks, F.P., "The Mythical Man-Month," Datamation, Vol. 20, No. 12, pp. 45-52, Dec. 1974.

5. Caine, S.H., and Gordon, E.K., "PDL--A Tool for Software Design", Program Design Language Reference Guide, Caine, Farber and Gordon, Inc., Pasadena, Ca., Sept. 18, 1974.

6. Constantine, L.L., Fundamentals of Program Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

7. Dahl, O.J., and Hoare, C.A.R., "Hierarchical Program Structures," Structured Programming, Academic Press, New York, 1972.

8. Dijkstra, E.W., "Notes on Structured Programming", Structured Programming, Academic Press, New York, 1972.

9. -----, "Structured Programming," Software Engineering Techniques, NATO Science Committee, Edited by J. Burton, and B. Randall, pp. 88-93, 1969

10. Gray, M., Landon, K., Documentation Standards, Brandon Systems Press, Inc., NY, 1969.

11. Hoare, C.A.R., "Notes on Data Structuring", Structured Programming, Academic Press, New York, 1972.

12. Jensen, K., and Wirth, N., Pascal User Manual and Report 2nd ed., Springer-Verlag, NY, 1974.

13. Katzan, H., Jr., Advanced Programming, D. VanNostrand Reinhold Co., NJ, 1970, pp. 152-163.

14. Kernighan, B.W., and Plauger, P.J., The Elements of Programming Style, McGraw-Hill Book Co., New York, 1974, pp.36-39.

15. Kiviat, P.J., Villanueva, R., and Markowitz, H., The SIMSCRIPT II Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1969.

16. Kleine, H., and Morris, R.V., "Modern Programming: A Definition", SIGPLAN Notices, Vol. 9, No. 9, Sept. 1974, pp. 14-17.

17. ----, Software Design and Documentation Language, JPL Publication 77-24, National Aeronautics and Space Administration, Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena Ca., 91103, Aug. 1977.

18. -----, Automating the Software Design Process by Means of the SDDL", Proceedings of the No. 15 Design Automation Conference, IEEE Catalog 378, Ch. 1363-1C, Las Vegas, Nev., June 1978, 371-379.

19. -----,"A Vehicle for Developing Standards for Simulation Programming", Proceedings of Winter ZZ Simulation Conference, Highland, Sargent, and Schmidt, eds., pp. 731-741.

20. Liskov, B., and Zilles, S., "Programming With Abstract Data Types', SIGPLAN Notices, March 1974, pp. 50-59.

21. Miller, E.F., Jr., A Compendium of Language Extensions to Support Structured Programming, RN-42, General Research Corp., Santa Barbara, Ca., Jan. 1973.

22. Mills, H.D.,"Top-Down Programming in Large Systems", in Debugging Techniques in Large Systems, edited by R. Rustin, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971, pp.43-45.

23. Myers, G.J., Composite Design: The Design of Modular Programs, Technical Report TR00.2406, IBM, PoughKeepsie, NY, Jan. 29, 1973.

24. Ogdin, C.A., Software Design for Microcomputers, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

25. Peters, L.J., Software Design: Method and Techniques, Yourdon Press, NY, 1981.

26. Schneider, G.M., Weingart, S.W., and Perlman, D.M., An Introduction to Programming and Problem Solving With Pascal, John Wiley, NY, 1978.

27. Tausworthe, R.C., Standardized Development of Computer Software, Part I Methods, Jet Propulsion Laboratory, Pasadena, Ca., 1976.

28. -----, Standardized Development of Computer Software. Part II Standards, Jet Propulsion Laboratory, Pasadena, Ca., Aug. 1978.
29. Wirth, N., Systematic Programming, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

30. Yourdon, E., Structured Walkthroughs, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

31. -----, Techniques of Program Structure and Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.

32. Zaks, R., Introduction to PASCAL, P310, SYBEX, Berkeley, Ca., 1980.

33. HIPO - A Design Aid and Documentation Technique, IBM Corp., Manual No. GC20-1851, White Plains, NY, IBM Data Processing Div., 1974.

```
**************************************************************
*                                                            *
*  SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE (SDDL)         *
*                                                            *
* An illustration of the application of SDDL using the      *
*  SDDL processor itself as the object of the example        *
*                                                            *
**************************************************************
```

PAGE LINE TABLE OF CONTENTS PAGE I

```
28 PROGRAM OBJECTIVES
29   ************************************************************************
30   * THE OBJECTIVE OF SDDL IS TO PROVIDE AN EFFECTIVE COMMUNICATIONS MEDIUM*
31   * TO SUPPORT THE DESCRIPTION AND DOCUMENTATION OF COMPLEX SOFTWARE      *
32   * SYSTEMS.  THIS OBJECTIVE IS MET BY PROVIDING:                        *
33   *                                                                      *
34   *    1. A DESCRIPTION LANGUAGE WITH FORMS AND SYNTAX THAT ARE SIMPLE,  *
35   *       UNRESTRICTIVE AND COMMUNICATIVE,                               *
36   *                                                                      *
37   *    2. A PROCESSOR WHICH CAN CONVERT DESIGN SPECIFICATIONS INTO AN    *
38   *       INTELLIGIBLE AND INFORMATIVE MACHINE REPRODUCIBLE DOCUMENT,     *
39   *                                                                      *
40   *    3. METHODOLOGY FOR EFFECTIVE USE OF THE LANGUAGE AND THE PROCESSOR.*
41   *                                                                      *
42   ************************************************************************
43 ENDPROGRAM
```

```
**************************************************************
*                                                            *
* This section exemplifies the use of SDDL to present a      *
*    "HIERARCHICAL INPUT - PROCESS - OUTPUT" (HIPO)          *
*            description of the SDDL system                  *
*                                                            *
**************************************************************
```

```
52 PROGRAM HIPO_DEVELOPMENT FOR THE SDDL PROCESSOR
53   ************************************************************************
54   *                                                                      *
55   * THE SDDL PROGRAM IS IMPLEMENTED IN TWO PASSES.  THIS MODULE DESCRIBES *
56   * THE TOP LEVEL OF THE PROGRAM WHICH DOES THE INITIAL SET UP AND INVOKES*
57   * THE TWO PASSES TO COMPLETE THE PROCESSING.                           *
58   *                                                                      *
59   ************************************************************************
60
61   INPUT:
62     [1]_SDDL_INVOCATION_COMMAND
63     [2]_SDDL_OUTPUT_FILE
64
65   PROCESS:
66              INITIALIZE THE SDDL PROCESSOR
67     [1]---->OPEN I/O AND SCRATCH FILES
68              EXECUTE HIPO_SDDL_FIRST_PASS---------------------------->(  4)
69              EXECUTE HIPO_SDDL_SECOND_PASS--------------------------->(  4)
70              APPEND EXECUTION SUMMARY DATA TO SDDL OUTPUT FILE --->      [2]
71              DELETE SCRATCH FILES
72              TERMINATE THE PROCESSOR
73
74   OUTPUT:
75     [2]_SDDL_OUTPUT_FILE
76
77 ENDPROGRAM
```

```
78 PROGRAM HIPO_SDDL_FIRST_PASS
79   ************************************************************************
80   *                                                                      *
81   * THE SDDL INPUT FILE IS READ, THE INPUT IS FORMATTED AND WRITTEN TO THE*
82   * SCRATCH FILE, AND CROSS REFERENCE AND SUMMARY DATA ARE COLLECTED.     *
83   *                                                                      *
84   ************************************************************************
85
86   INPUT:
87     [3]_SOURCE_DATA_FILE
88
89   PROCESS:
90     [3]---->CONVERT SOURCE DATA TO STRUCTURED FORMAT --->                [4]
91              DEVELOP TABLE OF CONTENTS DATA --->                        [5]
92              CAPTURE FORWARD REFERENCES DATA --->                       [6]
93              CAPTURE CROSS REFERENCE TABLE DATA --->                    [4]
94
95   OUTPUT:
96     [4]_DOCUMENT_SCRATCH_FILE
97     [5]_TABLE_OF_CONTENTS_FILE
98     [6]_FORWARD_REFERENCE_SCRATCH_FILE
99
00 ENDPROGRAM



02 PROGRAM HIPO_SDDL_SECOND_PASS
03   ************************************************************************
04   *                                                                      *
05   * THE TABLE OF CONTENTS IS WRITTEN, FORWARD REFERENCES ARE MERGED WITH  *
06   * THE BODY OF THE DOCUMENT AND THE CROSS REFERENCE TABLES ARE WRITTEN   *
07   *                                                                      *
08   ************************************************************************
09
10   INPUT:
11     [4]_DOCUMENT_SCRATCH_FILE
12     [5]_TABLE_OF_CONTENTS_FILE
13     [6]_FORWARD_REFERENCE_SCRATCH_FILE
14
15   PROCESS:
16     [5]---->WRITE TABLE OF CONTENTS --->                               [2]
17     [6],[4]->ADD MISSING PAGE REFS TO MODULE CALL STATEMENTS --->      [2]
18     [4]---->WRITE SDDL OUTPUT FILE --->                               [2]
19
20   OUTPUT:
21     [2]_SDDL_OUTPUT_FILE
22
23 ENDPROGRAM
```

```
*****************************************************************************
*                                                                           *
* This section exemplifies the use of SDDL for software design development *
*                                                                           *
*****************************************************************************
```

```
130 PROGRAM DESIGN_DATA_STRUCTURE AND GLOSSARY
131   ************************************************************************
132   *                                                                      *
133   * DATA STRUCTURES USED BY THE SDDL PROCESSOR ARE DEFINED AND EXPLAINED *
134   *                                                                      *
135   ************************************************************************
136
137   tb.INPUT.TEXT.BUFFER            GLOBAL CHARACTER ARRAY CONTAINING A
138                                   SINGLE STATEMENT FORMED BY CONCATEN-
139                                   ATION OF CONTINUED INPUT LINES
140
141   tb.TEXT.LENGTH                  LENGTH OF THE CURRENT INPUT STMT
142
143   LIST:  dx.TOKEN.DICTIONARY      LINKED LIST OF DICTIONARY ENTRIES
144
145     MEMBER ENTITY: dx.ENTRY         SINGLE DICTIONARY ENTRY
146       dx.CHARACTER.COUNT              TEXT LENGTH
147       dx.TEXT.POINTER                 POINTER TO ACTUAL TEXT FOR THIS ENTRY
148       dx.PROGRAM.NAME                 IF ENTRY IS A KEYWORD THEN PROGRAM NAME
149                                       ,ELSE NULL
150
151       LIST: dx.REFERENCES.TO.TOKEN  LIST OF ALL REFERENCES TO THE TOKEN
152         MEMBER ENTITY: dx.REFERENCE   SPECIFIC REFERENCE TO THE TOKEN
153           dx.PAGE.NUMBER                REFERENCE PAGE NUMBER
154           dx.LINE.NUMBER                REFERENCE LINE NUMBER
156
157
158   LIST:  ms.MODULE.STACK          PUSH-DOWN STACK OF NODES REPRESENTING
159                                     NESTED, CURRENTLY OPEN STRUCTURES
160
161     MEMBER ENTITY:  mx.NODE       OPEN STRUCTURE NODE
162       ms.INDENTATION.COLUMN         STARTING PRINT COLUMN FOR THIS NODE
163       ms.STRUCTURE.ID               IDENTITY OF ASSOCIATED STRUCTURE PARTS
164
165 ENDPROGRAM DATA STRUCTURE
```

```
 166 PROGRAM DESIGN_MAIN_DRIVER
 167   ********************************************************************************
 168   *                                                                              *
 169   * THE TOP LEVEL OF THE SDDL PROCESSOR IS SPECIFIED IN PSEUDO CODE              *
 170   *                                                                              *
 171   ********************************************************************************
 172
 173   FIRST: INITIALIZE THE PROGRAM
 174     ESTABLISH INITIAL VALUES FOR ALL PROGRAM VARIABLES
 175     SET UP DEFAULT STRUCTURES
 176     OPEN I/O AND SCRATCH FILES
 177
 178   NEXT:  PROCESS THE USER'S SOURCE STATEMENTS
 179     CALL DESIGN_FIRST_PASS----------------------------------------->(  8)
 180
 181   NEXT:  PRODUCE THE DOCUMENT SUMMARIES
 182     PREPARE THE MODULE REFERENCE TREE
 183     PREPARE THE MODULE CROSS REFERENCE TABLE
 184     PREPARE THE USER DEFINED CROSS REFERENCE TABLES
 185     PRINT THE TABLE OF CONTENTS
 186
 187   NEXT: PERFORM THE SECOND PASS OPERATIONS
 188     CALL DESIGN_SECOND_PASS  TO MERGE TEXT BODY WITH THE FORWARD REFERENCE
&         PAGE NUMBERS------------------------------------------------->(  9)
 189     PRINT THE SDDL EXECUTION STATISTICS
 190
 191 ENDPROGRAM
```

```
192 PROCEDURE DESIGN_FIRST_PASS
193   ************************************************************************
194   *                                                                      *
195   * SOURCE DATA IS READ AND FORMATTED ONTO A SCRATCH FILE                *
196   *                                                                      *
197   ************************************************************************
198
199   LOOP UNTIL ALL STATEMENTS IN [3]_SOURCE_DATA_FILE HAVE BEEN PROCESSED
200     CALL DESIGN_GET_NEXT_STATEMENT----------------------------------->(  9)
201     CALL DESIGN_TOKEN.FINDER (FINDS THE FIRST TOKEN IN THE STATEMENT)>(   )
202     IF dx.TOKEN.TYPE IS AN "IDENTIFIER"
203       CALL DESIGN_TOKEN.DICTIONARY.SEARCH---------------------------->(   )
204     ENDIF
205     IF THE TOKEN WAS FOUND AND IT IS A KEYWORD
206       CALL DESIGN_KEYWORD.STATEMENT.PROCESSOR------------------------>(   )
207
208     ELSE  THE STATEMENT DOES NOT BEGIN WITH A KEYWORD
209       IF THE ms.MODULE.STACK IS EMPTY
210         PUSH A DUMMY MODULE ONTO THE ms.MODULE.STACK
211       ENDIF
212
213       CALL DESIGN_SOURCE.LISTER TO WRITE STMT------------------------>(   )
214     ENDIF
215
216     FLUSH ANY "ERROR MESSAGES" TRIGGERED BY THE STATEMENT
217
218   REPEAT
219 ENDPROCEDURE
```

```
220 PROCEDURE DESIGN_SECOND_PASS
221   ************************************************************************
222   *                                                                      *
223   * AFTER THE USER'S INPUT TEXT HAS BEEN PROCESSED THIS MODULE MERGES    *
224   * THE FORWARD REFERENCES INTO THE BODY OF THE TEXT AND WRITES IT TO     *
225   * THE FINAL OUTPUT FILE                                                *
226   *                                                                      *
227   ************************************************************************
228
229   FIRST:  MERGE THE FORWARD REFERENCES INTO THE BODY OF THE TEXT
230     REWIND [4]_DOCUMENT_SCRATCH_FILE AND OPEN IT FOR INPUT
231     REWIND [6]_FORWARD_REFERENCE_SCRATCH_FILE AND OPEN FOR INPUT
232     READ THE DATA FOR THE FIRST FORWARD REFERENCE
233     LOOP UNTIL ALL LINES IN [4]_DOCUMENT_SCRATCH_FILE HAVE BEEN PROCESSED
234       READ NEXT INPUT LINE
235       IF THE LINE REQUIRES A FORWARD REFERENCE LINE NUMBER
236         ADD THE REQUIRED INFORMATION TO THE INPUT LINE
237         READ THE DATA FOR THE NEXT FORWARD REFERENCE
238       ENDIF
239       PRINT THE LINE TO [2]_SDDL_OUTPUT_FILE
240     REPEAT
241
242   NEXT: PRINT THE REMAINING DOCUMENT SUMMMARIES TO [2]_SDDL_OUTPUT_FILE
243     PRINT THE MODULE REFERENCE TREE
244     PRINT THE MODULE CROSS REFERENCE TABLE
245     PRINT THE USER SPECIFIED CROSS REFERENCE TABLES
246
247 ENDPROCEDURE



249 PROCEDURE DESIGN_GET_NEXT_STATEMENT
250   ************************************************************************
251   *                                                                      *
252   * INPUT LINES ARE READ AND IF LINE CONTINUATION IS INDICATED THE LINES *
253   * ARE CONCATENATED INTO A SINGLE STATEMENT.  STATEMENT PARAMETERS ARE  *
254   * ESTABLISHED                                                          *
255   *                                                                      *
256   ************************************************************************
257
258   FIRST: GET THE FIRST INPUT LINE
259     READ THE INPUT LINE INTO THE tb.INPUT.TEXT.BUFFER
260
261   NEXT: CHECK FOR INPUT LINE CONTINUATION
262     LOOP UNTIL THE STATEMENT IS COMPLETE
263       FIND THE LAST NON BLANK CHARACTER OF THE LINE
264       IF THE CHARACTER IS THE CONTINUATION MARK
265         READ THE NEXT INPUT LINE AND ADD IT TO THE tb.INPUT.TEXT.BUFFER
266       ELSE
267     <---EXITLOOP
268         ENDIF
269     REPEAT
270
271   NEXT: SET tb.TEXT.LENGTH
272
273 ENDPROCEDURE
```

```
*************************************************************
*                                                           *
* This section exemplifies the use of SDDL to process       *
*          the pascal implementation of SDDL                *
*                                                           *
*************************************************************
```

```
{PROGRAM  CODE_MAIN}
  {
  ************************************************************************
  *                                                                      *
  * THE TOP LEVEL DRIVER OF THE SDDL PROCESSOR IS SPECIFIED IN PSEUDO CODE*
  *                                                                      *
  ************************************************************************
  }
  BEGIN
    {FIRST: INITIALIZE THE PROGRAM}
      {CALL} Initialization;-------------------------------------->(    )

    {NEXT:  PROCESS THE USER'S SOURCE STATEMENTS}
      IF  NOT  MoreData  THEN    {                                REV %16 }
        WRITELN (Output, 'NO INPUT FOR THIS RUN')
      ELSE
        BEGIN                                 {                   REV %17 }
          {CALL} CODE_FirstPass;-------------------------------->( 12)
          {FIRST: DEVELOP DOCUMENT SUMMARIES}
            {CALL} ProduceInvocationTree;------------------------>(    )
            {CALL} ProduceXrefTables;---------------------------->(    )
            {CALL} ProduceTableOfContents;------------------------>(    )

          {NEXT:  PERFORM THE SECOND PASS OPERATIONS}
            {CALL} CODE_SecondPass;------------------------------->( 14)
            {CALL} EndSummary;------------------------------------>(    )
        END
      {ENDIF}
  END
{ENDPROGRAM}
```

```
314 PROCEDURE CODE_FirstPass;
315 {
316 ************************************************************************
317 *                                                                      *
318 * SOURCE DATA IS READ AND FORMATTED ONTO A SCRATCH FILE                *
319 *                                                                      *
320 ************************************************************************
321 }
322 VAR Keyword: KeywordSelector;
323
324 BEGIN
325   WHILE StmtFound DO
326   BEGIN
327     IF TokenType >= IdentifierToken THEN
328       BEGIN
329         {CALL} LookupKeyword (FALSE, Keyword);----------------------->(    )
330         IF Keyword <> NIL THEN
331           {CALL} ProcessKeywordStatement (Keyword)------------------->(    )
332         ELSE
333           {CALL} ProcessPassiveStatement----------------------------->(    )
334         {ENDIF}
335       END
336     ELSE
337       {CALL} CODE_GetNextStatement------------------------------------>( 13)
338     {ENDIF}
339   END;
340   {CALL} ReduceStack(0)--------------------------------------------->(    )
341 END;
```

```
342 PROCEDURE CODE_GetNextStatement;
343 {
344 ************************************************************************
345 *                                                                      *
346 * INPUT LINES ARE READ AND IF LINE CONTINUATION IS INDICATED THE LINES *
347 * ARE CONCATENATED INTO A SINGLE STATEMENT.  STATEMENT PARAMETERS ARE  *
348 * ESTABLISHED                                                          *
349 *                                                                      *
350 ************************************************************************
351 }
352 VAR I: INTEGER;
353     Ch: CHAR;
354
355 BEGIN
356   {FIRST: GET THE NEXT LINE OF INPUT}
357     {
358     ************************************************************************
359     * CODE BODY OMITTED                                                    *
360     ************************************************************************
361     }
362
363   {NEXT:  CHECK FOR INPUT LINE CONTINUATION}
364     {
365     ************************************************************************
366     * CODE BODY OMITTED                                                    *
367     ************************************************************************
368     }
369
370   {NEXT:  SET tb.TEXT.LENGTH  AND RETRIEVE FIRST TOKEN OF THE STMT}
371     {
372     ************************************************************************
373     * CODE BODY OMITTED                                                    *
374     ************************************************************************
375     }
376 END;
```

```
377 PROCEDURE CODE_SecondPass;
378 {
379 ************************************************************************
380 *                                                                      *
381 * AFTER THE USER'S INPUT TEXT HAS BEEN PROCESSED THIS MODULE MERGES    *
382 * THE FORWARD REFERENCES INTO THE BODY OF THE TEXT AND WRITES IT TO     *
383 * THE FINAL OUTPUT FILE                                                *
384 *                                                                      *
385 ************************************************************************
386 }
387 VAR Reference: ForwrdRefEntry;
388     I, Length, LineCount: INTEGER;
389
390 BEGIN
391   {FIRST: MERGE THE FORWARD REFERENCES INTO THE BODY OF THE TEXT}
392     WITH Reference DO
393     BEGIN
394       RESET (Scratch);
395       RESET (ForwardReferences);
396       {
397       ************************************************************************
398       * CODE BODY OMITTED                                                    *
399       ************************************************************************
400       }
401     END
402 END;
```

```
*****************************************************************
*                                                               *
* This section exemplifies the use of SDDL to describe the      *
* command for invoking the SDDL processor on the VAX 11/780     *
*                                                               *
*****************************************************************
```

```
414 SDDL PROGRAM_INVOCATION_FORMAT_SPECIFICATION
415   ************************************************************************
416   *                                                                      *
417   * THE STEPS FOR INVOCATION OF THE SDDL PROCESSOR AND FOR               *
418   * NAMING THE INPUT AND OUTPUT FILES ARE PRESENTED BELOW                *
419   *                                                                      *
420   ************************************************************************
421
422   FIRST: Enter the SDDL invocation command
423 <---ENTER LITERAL: @SDDL
424
425   OPTIONAL NEXT:  Start a new input line
426 <---ENTER:  CARRIAGE RETURN
427     VAX RESPONSE:
428     >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
429     >   ENTER THE INPUT FILE NAME                                        >
430     >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
431
432   NEXT:  Specify the input file name
433 <---ENTER THE NAME OF THE INPUT FILE
434
435   NOTE:
436     IF THE FILE NAME EXTENSION IS OMITTED
437       DEFAULT EXTENSION  "SDD"  WILL BE APPLIED
438     ENDIF
439
440   NOTE:
441     IF THE INPUT FILE SPECIFIED DOES NOT EXIST
442       VAX RESPONSE:
443       >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
444       >   FILE NAMED DOES NOT EXIST, TRY AGAIN                           >
445       >   ENTER THE INPUT FILE NAME                                      >
446       >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
447       COMMAND MACRO RETURNS TO PREVIOUS STEP
448     ENDIF
449
450   OPTIONAL NEXT:  Start a new input line
451 <---ENTER:  CARRIAGE RETURN
452     VAX RESPONSE:
453     >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
454     >   ENTER THE OUTPUT FILE NAME                                       >
455     >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
456
457   NEXT:  Specify the name of the output file
458 <---ENTER THE NAME OF THE OUTPUT FILE OR CARRIAGE RETURN
459     IF A CARRIAGE RETURN IS ENTERED
460
461       "LIS" WILL BE APPENDED TO THE INPUT FILE NAME TO FORM THE
462       NAME OF THE OUTPUT FILE
463
464     ELSE IF THE EXTENSION NAME IS OMITTED
465
466       "LIS" WILL BE APPENDED TO THE OUTPUT NAME SUPPLIED
467
468     ELSE
469       THE OUTPUT FILE NAME WILL BE USED AS SPECIFIED
470     ENDIF
```

```
  NEXT:  SDDL processor is executed

     VAX RESPONSE:
     >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
     >                                                                    >
     >   INPUT/OUTPUT FILES ARE "INPUT" & "OUTPUT"                        >
     >                                                                    >
     >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

     THE SDDL PROGRAM IS EXECUTED

     THE CONSOLE BELL IS RUNG TO NOTIFY USER

END
```

```
**********************************************
*                                            *
*                 TOP LEVEL                  *
*                                            *
*           SYNTAX SPECIFICATION             *
*                                            *
*                  FOR THE                   *
*                                            *
* SOFTWARE DESIGN & DOCUMENTATION LANGUAGE   *
*                                            *
**********************************************
```

```
509 SDDL_CONSTRUCT: SDDL-PROGRAM
510
511   FIRST: The optional output suppression control
512     SEE SUPPRESS.OUTPUT.DIRECTIVE---------------------------------->(    )
513
514   NEXT: The document
515     ITERATION:
516       SELECTION:
517         * TITLE-PAGE---------------------------------------------->( 19)
518       OR
519         * MODULE-GROUP-------------------------------------------->( 20)
520       OR
521         * FORMAT-CONTROL-DIRECTIVE-------------------------------->(    )
522       END_SELECTION_OPTIONS
523     END_ITERATION
524
526 END_CONSTRUCT



528 SDDL_CONSTRUCT: TITLE-PAGE
529
530 *FIRST TITLE.INITIATOR.DIRECTIVE
531   FIRST: The DIRECTIVE keyword
532     LITERAL: #TITLE
533   OPTIONAL NEXT: The character to be used to enclose the text body
534     SELECTION:
535       Any PUNCTUATION, MARK, STRING, or COMMENT character
536     OR:
537       NULL (Default = LITERAL: *)
538     END_SELECTION_OPTIONS
539   NEXT: The title of the Title Page for entry into the Table of Contents
540     Any text
541   END_STATEMENT
542
543 *NEXT: TITLE-PAGE-BODY
544   Up to a full page of statements,  none of which may begin with #END
545   (Note that these are lines, not statements)
546   (This therefore precludes the use of line continuation)
547
548 *LAST: TITLE.BOX.TERMINATOR.DIRECTIVE
549   FIRST:
550     LITERAL: #END
551   OPTIONAL NEXT:
552     Any additional notation (ignored)
553   END_STATEMENT
554
556 END_CONSTRUCT
```

```
557 SDDL_CONSTRUCT: MODULE-GROUP
558
559 *FIRST MODULE.INITIATOR.STATEMENT
560   FIRST: The keyword for a MODULE INITIATOR
561     SEE BUILT.IN.KEYWORDS---------------------------------------->(    )
562   OPTIONAL NEXT:  Noise word
563     SELECTION:
564       LITERAL: FOR
565     OR
566       LITERAL: TO
567     OR
568       * PUNCTUATION------------------------------------------------>(    )
569     END_SELECTION_OPTIONS
570   NEXT: The name of the MODULE
571     * IDENTIFIER-------------------------------------------------->(    )
572   OPTIONAL NEXT:
573     Any text may be added to complete the statement
574   END_STATEMENT
575
576 *NEXT: MODULE-GROUP-BODY
577   ITERATION:
578     SELECTION:
579       * PASSIVE.STATEMENT------------------------------------------>(    )
580     OR
581       * MODULE.INVOCATION.STATEMENT--------------------------------->(    )
582     OR
583       * ESCAPE.STATEMENT for the extant MODULE---------------------->(    )
584     OR
585       * MODULE.SUBSTRUCTURE.STATEMENT for the extant MODULE--------->(    )
586     OR
587       * TEXT-BOX-GROUP---------------------------------------------->(    )
588     OR
589       * FORMAT-CONTROL-DIRECTIVE------------------------------------>(    )
590     OR
591       * BLOCK-GROUP------------------------------------------------->(    )
592     END_SELECTION_OPTIONS
593   END_ITERATION
594
595 *NEXT: MODULE.TERMINATOR.STATEMENT
596   FIRST: The TERMINATOR keyword corresponding to the INITIATOR keyword
597     SEE BUILT.IN.KEYWORDS---------------------------------------->(    )
598   NEXT:
599     Any text may be added to complete the statement
600   END_STATEMENT
601
603
604 END_CONSTRUCT
```

```
LINE  PAGE     ******   MODULE INVOCATION TREE   ******                PAGE  21

   1     1    OBJECTIVES

   2     3    HIPO_DEVELOPMENT
   3     4    .  HIPO_SDDL_FIRST_PASS
   4     4    .  HIPO_SDDL_SECOND_PASS

   5     6    DESIGN_DATA_STRUCTURE

   6     7    DESIGN_MAIN_DRIVER
   7     8    .  DESIGN_FIRST_PASS
   8     9    .  :  DESIGN_GET_NEXT_STATEMENT
   9     *    .  :  DESIGN_TOKEN.FINDER
  10     *    .  :  DESIGN_TOKEN.DICTIONARY.SEARCH
  11     *    .  :  DESIGN_KEYWORD.STATEMENT.PROCESSOR
  12     *    .  :  DESIGN_SOURCE.LISTER
  13     9    .  DESIGN_SECOND_PASS

  14    11    CODE_MAIN
  15     *    .  Initialization
  16    12    .  CODE_FirstPass
  17     *    .  :  LookupKeyword
  18     *    .  :  ProcessKeywordStatement
  19     *    .  :  ProcessPassiveStatement
  20    13    .  :  CODE_GetNextStatement
  21     *    .  :  ReduceStack
  22     *    .  ProduceInvocationTree
  23     *    .  ProduceXrefTables
  24     *    .  ProduceTableOfContents
  25    14    .  CODE_SecondPass
  26     *    .  EndSummary

  27    16    PROGRAM_INVOCATION_FORMAT_SPECIFICATION

  28    19    SDDL-PROGRAM
  29     *    .  SUPPRESS.OUTPUT.DIRECTIVE
  30    19    .  TITLE-PAGE
  31    20    .  MODULE-GROUP
  32     *    .  FORMAT-CONTROL-DIRECTIVE

  33    19    TITLE.INITIATOR.DIRECTIVE

  34    19    TITLE-PAGE-BODY

  35    19    TITLE.BOX.TERMINATOR.DIRECTIVE

  36    20    MODULE.INITIATOR.STATEMENT
  37     *    .  BUILT.IN.KEYWORDS
  38     *    .  PUNCTUATION
  39     *    .  IDENTIFIER

  40    20    MODULE-GROUP-BODY
  41     *    .  PASSIVE.STATEMENT
  42     *    .  MODULE.INVOCATION.STATEMENT
  43     *    .  ESCAPE.STATEMENT
  44     *    .  MODULE.SUBSTRUCTURE.STATEMENT
  45     *    .  TEXT-BOX-GROUP
  46     *    .  FORMAT-CONTROL-DIRECTIVE
  47     *    .  BLOCK-GROUP
```

MODULE
CROSS REFERENCE LISTING PAGE 23

```
BLOCK-GROUP
    PAGE  20  *NEXT: MODULE-GROUP-BODY                    591
BUILT.IN.KEYWORDS
    PAGE  20  *FIRST MODULE.INITIATOR.STATEMENT           561
    PAGE  20  *NEXT: MODULE.TERMINATOR.STATEMENT          597
CODE_FirstPass
    PAGE  11  {PROGRAM  CODE_MAIN                         301
    PAGE  12  PROCEDURE CODE_FirstPass                    314
CODE_GetNextStatement
    PAGE  12  PROCEDURE CODE_FirstPass                    337
    PAGE  13  PROCEDURE CODE_GetNextStatement             342
CODE_MAIN
    PAGE  11  {PROGRAM  CODE_MAIN                         284
CODE_SecondPass
    PAGE  11  {PROGRAM  CODE_MAIN                         308
    PAGE  14  PROCEDURE CODE_SecondPass                   377
DESIGN_DATA_STRUCTURE
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE               130
DESIGN_FIRST_PASS
    PAGE   7  PROGRAM DESIGN_MAIN_DRIVER                  179
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                 192
DESIGN_GET_NEXT_STATEMENT
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                 200
    PAGE   9  PROCEDURE DESIGN_GET_NEXT_STATEMENT         249
DESIGN_KEYWORD.STATEMENT.PROCESSOR
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                 206
DESIGN_MAIN_DRIVER
    PAGE   7  PROGRAM DESIGN_MAIN_DRIVER                  166
DESIGN_SECOND_PASS
    PAGE   7  PROGRAM DESIGN_MAIN_DRIVER                  188
    PAGE   9  PROCEDURE DESIGN_SECOND_PASS                220
DESIGN_SOURCE.LISTER
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                 213
DESIGN_TOKEN.DICTIONARY.SEARCH
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                 203
DESIGN_TOKEN.FINDER
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                 201
EndSummary
    PAGE  11  {PROGRAM  CODE_MAIN                         309
ESCAPE.STATEMENT
    PAGE  20  *NEXT: MODULE-GROUP-BODY                    583
FORMAT-CONTROL-DIRECTIVE
    PAGE  19  SDDL_CONSTRUCT: SDDL-PROGRAM                521
    PAGE  20  *NEXT: MODULE-GROUP-BODY                    589
HIPO_DEVELOPMENT
    PAGE   3  PROGRAM HIPO_DEVELOPMENT                     52
HIPO_SDDL_FIRST_PASS
    PAGE   3  PROGRAM HIPO_DEVELOPMENT                     68
    PAGE   4  PROGRAM HIPO_SDDL_FIRST_PASS                 78
HIPO_SDDL_SECOND_PASS
    PAGE   3  PROGRAM HIPO_DEVELOPMENT                     69
    PAGE   4  PROGRAM HIPO_SDDL_SECOND_PASS               102
IDENTIFIER
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                 202
    PAGE  20  *FIRST MODULE.INITIATOR.STATEMENT           571
```

HIPO DATA SETS
CROSS REFERENCE LISTING PAGE 25

++

```
[1]
    PAGE    3  PROGRAM HIPO_DEVELOPMENT                    67
[1]_SDDL_INVOCATION_COMMAND
    PAGE    3  PROGRAM HIPO_DEVELOPMENT                    62
[2]
    PAGE    3  PROGRAM HIPO_DEVELOPMENT                    70
    PAGE    4  PROGRAM HIPO_SDDL_SECOND_PASS              116  117  118
[2]_SDDL_OUTPUT_FILE
    PAGE    3  PROGRAM HIPO_DEVELOPMENT                    63   75
    PAGE    4  PROGRAM HIPO_SDDL_SECOND_PASS              121
    PAGE    9  PROCEDURE DESIGN_SECOND_PASS               239  242
[3]
    PAGE    4  PROGRAM HIPO_SDDL_FIRST_PASS                90
[3]_SOURCE_DATA_FILE
    PAGE    4  PROGRAM HIPO_SDDL_FIRST_PASS                87
    PAGE    8  PROCEDURE DESIGN_FIRST_PASS                199
[4]
    PAGE    4  PROGRAM HIPO_SDDL_FIRST_PASS                90   93
    PAGE    4  PROGRAM HIPO_SDDL_SECOND_PASS              117  118
[4]_DOCUMENT_SCRATCH_FILE
    PAGE    4  PROGRAM HIPO_SDDL_FIRST_PASS                96
    PAGE    4  PROGRAM HIPO_SDDL_SECOND_PASS              111
    PAGE    9  PROCEDURE DESIGN_SECOND_PASS               230  233
[5]
    PAGE    4  PROGRAM HIPO_SDDL_FIRST_PASS                91
    PAGE    4  PROGRAM HIPO_SDDL_SECOND_PASS              116
[5]_TABLE_OF_CONTENTS_FILE
    PAGE    4  PROGRAM HIPO_SDDL_FIRST_PASS                97
    PAGE    4  PROGRAM HIPO_SDDL_SECOND_PASS              112
[6]
    PAGE    4  PROGRAM HIPO_SDDL_FIRST_PASS                92
    PAGE    4  PROGRAM HIPO_SDDL_SECOND_PASS              117
[6]_FORWARD_REFERENCE_SCRATCH_FILE
    PAGE    4  PROGRAM HIPO_SDDL_FIRST_PASS                98
    PAGE    4  PROGRAM HIPO_SDDL_SECOND_PASS              113
    PAGE    9  PROCEDURE DESIGN_SECOND_PASS               231
```

DATA ITEMS
CROSS REFERENCE LISTING PAGE 26

```
dx.CHARACTER.COUNT
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              146
dx.ENTRY
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              145
dx.LINE.NUMBER
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              154
dx.PAGE.NUMBER
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              153
dx.PROGRAM.NAME
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              148
dx.REFERENCE
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              152
dx.REFERENCES.TO.TOKEN
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              151
dx.TEXT.POINTER
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              147
dx.TOKEN.DICTIONARY
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              143
dx.TOKEN.TYPE
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                202
ERROR MESSAGES
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                216
ms.INDENTATION.COLUMN
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              162
ms.MODULE.STACK
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              158
    PAGE   8  PROCEDURE DESIGN_FIRST_PASS                209  210
ms.STRUCTURE.ID
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              163
nx.NODE
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              161
tb.INPUT.TEXT.BUFFER
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              137
    PAGE   9  PROCEDURE DESIGN_GET_NEXT_STATEMENT        259  265
tb.TEXT.LENGTH
    PAGE   6  PROGRAM DESIGN_DATA_STRUCTURE              141
    PAGE   9  PROCEDURE DESIGN_GET_NEXT_STATEMENT        271
    PAGE  13  PROCEDURE CODE_GetNextStatement            370
```

REVISIONS
CROSS REFERENCE LISTING PAGE 27

++

```
%16
     PAGE   11   (PROGRAM   CODE_MAIN                          297
%17
     PAGE   11   (PROGRAM   CODE_MAIN                          300
```

Modular Design of Real-Time Systems

V.H.Haase, TU Graz

1. Distributed Systems

This paper deals with the construction of software for distributed real-time systems. Both distribution and real-time are necessary elements of our consideration: neither "distribution-only" - systems as e.g. packet-switching networks nor "real-time-only" problems like interrupt-handlers or schedulers have the complexity of multi-micro-processor based process-control systems we are regarding.
Computing systems which are no more von-Neumann machines are used more and more frequently especially in process-control and automation systems. Academic computer science seems to ignore the fact that microprocessors applied in large control systems constitute a distributed multiprocessor machine which is no more sequential in behavior, and does not have a global system state. One field of application are SIMD (single - instruction-multiple data) machines like array-processors; we concentrate on MIMD (multiple-instruction-multiple data) systems with a large number of loosely-coupled processors working in parallel, each of them resembling a sequential (finite-state) machine. Typical application fields are the control of large machinery as e.g. in steel or chemical industry, of airplanes, trains, power plants etc. (Fig.1).

2. Software Construction

High level programming languages have been the most important tool for the construction of programs. They have been designed for the description of algorithms which are executed on single sequentially working (von-Neumann) processors. That means that actions have to be strictly ordered in time, concurrency is not provided.

While these features map sufficiently good onto the architecture of singleprocessor-machines this is no more valid for multiprocessors. We have to decide whether we add features (like synchronization, tasking) to sequential programming languages or if we decide to choose new structured methods for program construction.Doing this it is extremely important to find a good mapping of problem-structure onto software-structure, and of software-structure onto hardware structure. Real-time requirements can only be met efficiently if the structures are very similar (→ Fig.2).

3. Parallelism vs. Sequentialism

Many "modern"programming languages like ADA, CHILL, MODULA or PEARL contain elements for tasking and synchronization. Nevertheless they are in principle algorithmic languages for sequential machines. Parallelism is an "add upon" - feature, and in most cases also implementation is like that: by a sophisticated organization of the operating system parallelism is simulated on sequentially working processors.

This type of multiprogramming operating systems becomes very complex and unreliable if distribution and real-time requirements are present. As software costs surpass hardware costs it seems reasonable to map tasks 1:1 on microprocessors, and to have a similar correspondence between hardware-connections of various processors,and software synchronization.
Following this scheme each processor in the system ideally executes one sequential algorithm (in most cases a cyclic program); these programs are coordinated by features which constitute a global synchronization scheme. Not parallel features are incorporated in sequential programs, but rather sequential programs are building bricks in a parallel distributed construction process (Fig. 3).

4. How to use modules as "programmer's atoms"

We start from the following assumptions:

a) It is easy and well understood to write efficient and reliable sequential programs in high level languages.

b) Distributed systems use software which consists of sequential modules which are executed (partly) in parallel, and which have to be synchronized.

c) If we define a module as (the part of) a sequential program between two points of synchronization, modules can be used as bricks to build parallel programming systems, where software structure is homomorphic to hardware structure, and software synchronization is supported by hardware communication equipment.

In the PARC-method to be discussed here sequential and parallel program construction are strictly separated, only a small set of control statements is sufficient to describe the synchronization of sequential program modules, and semantics of parallel programs can be defined by predicate transformers. The method has been derived from Dijstras guarded commands [1] , has been developped by Halling and Haase [2, 3, 4] , and is related to Hoare's approach [5] and the OCCAM-Language [6].
The concept of guards (conditions) and actions is similar as in Petri Nets [7] .

A distributed programming system is composed out of a sequence of PARCS (parallel constructs). Each PARC describes a set of processes being executable on different (virtual) processors (Fig. 4) in a phase of the operation of the system.
A PARC is a collection of conditional actions

$$(\text{condition} \longrightarrow \text{action})^*$$

which can be executed in parallel. Actions are sequential programs (modules) which are started when the condition becomes true. Condition is a predicate on (a subset of) the state of the system. Actions may be marked as being executable only once (parallel-IF-style) or repeated (parallel-DO-style).

Syntax

```
program :: = parc*
parc :: =  module |
           PARC parc-name
           conditional-action { □ conditional action }*
           ENDPARC parc-name
```

conditional-action :: = condition → program

condition :: = Boolean expression

module :: = [global-sync] module-name [global-sync] REPEAT

Comments: "Boolean-expression" is defined over the state of the environment which consists of the values of global variables and of the occurence of communication signals. Global variables can only be altered, and communication signals issued in the optional "global-sync" parts of the actions. "Modules", which may or may not be REPEATed write only local variables, and are not allowed to address communication signals.

Example

```
   PARC  INIT
   STARTBUTTON ⟶ STARTTRANSPORT  (TRANSPORT-ON)
☐ STARTBUTTON AND TRANSPORT-ON ⟶   RESETPRESS 1
☐ STARTBUTTON AND TRANSPORT-ON ⟶   RESETPRESS 2
☐ STARTBUTTON AND TRANSPORT-ON ⟶   RESETCUTTER
☐ STARTBUTTON AND TRANSPORT-ON ⟶   RESETSPOOLER
   ENDPARC INIT

   PARC OPERATION

☐ TEMPERATURE > 1200 AND BLOCKENTERS_1
      AND TRANSPORT-ON ⟶ OPERATEPRESS_1 REPEAT
☐ BLOCKENTERS 2 AND TRANSPORT-ON ⟶ OPERATEPRESS_2 REPEAT
☐ BLOCKENTERS 3 AND TRANSPORT-ON ⟶ START_SPOOLER
☐ SPOOLER FULL ⟶ (SETSTOP) OPERATE_CUTTER
☐ SETSTOP ⟶ STOP-TRANSPORT (TRANSPORT-OFF)
  ENDPARC OPERATION
```

Semantics: Semantics of PARC is related to semantics of guarded commands as shown in [3] and [4]. As the syntax used in this paper does not distinguish between IF-PARCs and DO-PARCs - this is more useful for programming in the large - the semantic has to be derived from putting together IF and DO-rules.

If a PARC consists of a set of n repeated conditional-action and a set of m not repeated conditional actions:

```
        PARC name
        cond_1   ⟶ module_1
          .
          .
      □ cond_n   ⟶ module_n
      □ cond_1R  ⟶ module_1R REPEAT
          .
          .
      □ cond_mR  ⟶ module_mR REPEAT
        ENDPARC name
```

this is equivalent to:

```
        PARC  outer
              true ⟶ PARC inner-if
            □ true ⟶ PARC inner-do
        ENDPARC  outer
```

with:

```
        PARC  inner-if
        cond_1 ⟶ module_1
          .
          .
      □ cond_n ⟶ module_n
        ENDPARC  inner-if    (this is a conventional IF-PARC)
```

and:

PARC inner-do

$cond_{1R}$ → $module_{1R}$ REPEAT

.
.

□ $cond_{mR}$ → $module_{mR}$ REPEAT

ENDPARC inner-do (this is a conventional DO-PARC)

Predicate transformers for IF- and DO-parcs are equivalent to predicate-transformers for guarded command sets ([1], [3]). So the behavior of the whole system can be derived from the effects of the va ious-modules taking into account the guarding conditions. Correctness can strictly be derived in two steps: first the individual modules, afterwards the whole compound. This is the same sequence as in the construction process.

5. An Example

The PARC-method has been applied in a number of software-development projects mainly in research environments. The construction of a distributed operating system for a PDP-11 [8] will be discussed in some detail; the control program for a robot was implemented using PARCs on 8080 microprocessors; recently we use it as a paper and pencil method for programming in the large of telecommunication systems, e.g. multi-microcomputer-gateways for network interconnection.

The PDP-11 project consisted of the implementation of an operating system able to interpret PARCs, and of the use of this system in applications. The implementation is based on a virtual machine model using Concurrent Pascal with the SOLO Operating System on a PDP 11/45.

Our aim was the simulation of a distributed architecture by means of a high-level language with pararllel programming features. Pascal is used to describe one way in which PARCs can be interpreted in a system with any kind of supervisory processor. Hence some experimental experience can be gained with the concept on a high level of description. A second aim is the definition of a simulation model which allows the test of dedicated process control applications including the development of a special purpose operating system.

As a guideline we may quote Brinch Hansen [9]

"Eventually industry will be using complicated specialized networks of microprocessors. These dedicated computer systems may not be programmable in the sense that they can execute arbitrary programs. They may indeed owe their efficiency to fixed algorithms built into the hardware. But somebody must still write and verify these concurrent algorithms. It seems very attractive to write a concurrent program in an abstract language, test it on a minicomputer, and then derive the most straightforward multiprocessor architecture from the program itself."

The specification language structures are translated by a precompiler written in Pascal into a Concurrent Pascal target sytem. Starting with the specification given by the PARC control program we generate a dedicated system which consists of three parts (fig.5) :

- the Concurrent Pascal kernel, responsible for processor multiplexing, handling of monitor calls and i/o;
- a skeleton of system components, responsible for the execution of PARC-structures. This run-time system simulates a distributed architecture, and all synchronization and communication mechanism required by a control program;
- problem-dependent sections; e.g. guard functions, condition statement lists, and sequential modules.

The virtual machine built out of these parts simulates the distributed microcomputer architecture specified in the control program.

Although the monitor-concept of Concurrent Pascal is not ideal for a microcomputer network [10], the access graph of the virtual machine (Fig.6) reflects the main components of a distributed system interpreting the PARC-structures. A group of components working as the control unit of the system is responsible for the execution of PARCs, implying evaluation of guards, execution of entry and exit actions, and access to conditions, which are used by more than one control module ("links").

The group of guarded processes, each running as a Concurrent Pascal process, communicates via a shared data monitor. The two subsystems exchange start- and termination-signals via a monitor master.

This Concurrent Pascal system has been tested using various classical examples including the producer-consumer problem, and a process control application (mixing of chemicals according to given recipes).

The description of the implementation model by means of a high-level language turned out to be a clear, well-structured, easy-to-modify, and easy-to-test approach. On the other hand restrictions which the monitor concept imposes on the program design, the language overhead of Concurrent Pascal, and the poor program development tools of the SOLO operating system reduce the applicability of this implementation when solving "real" process control problems. It is therefore concluded that steps towards a more application oriented implementation should be made.

6. Conclusion

An application oriented design concept for parallel programs as well as a tentative implementation model was shown. It can be regarded as the prototype of a kind of automation systems where not only the application program but also the architecture of the system (both software, and - in future - also hardware) can be defined by the user. This could be done using a single application oriented description method. The key issue is the separation of sequential programs from synchronization mechanisms. This structure can easily be mapped on to a hardware architecture based on multiple microprocessors and a bus system. There is no necessity for complex real-time-operating-systems in the individual processors.

The Concurrent Pascal implementation described is an experimental system; it should be a model for a set of tools for the development of control systems. These tools will consist of precompilers which translate PARC-specifications into appropriate programming languages (not necessarily PASCAL, also FORTRAN and BASIC are feasible), of

"system-builder"-programs which analyse requirements and resources and suggest suitable configurations, and of operating-system skeletons (including device handlers, message protocols etc.). All tools may be implemented in software and/or firmware (ROMs).

We think that the approach to use sequential program-modules as they are, as building bricks for parallel and distributed systems is both a proper engineering as well as an economic solution.

Acknowledgement

I want to thank Horst Halling (Jülich) and Wolf-Michael Dehnert (München) for several ideas which have been included in this paper.

References

[1] E.W.Dijkstra: A Discipline of programming, Englewood Cliffs (1976)

[2] H.Halling, K.Bürger, H.Heer: Implementation and Application of PARCs. SOCOCO 79, Prague (1979)

[3] V.Haase: Specification and Construction of Real-Time-Programs with PARCs. Angewandte Informatik 5/80 (1980)

[4] V.Haase: Real-Time Behaviour of Programs: IEEE TSE 7, p. 494 (1981)

[5] C.A.R.Hoare: Communication Sequential Processes, CACM 21, p. 666 (1978)

[6] R.Taylor, P.Wilson: Process-oriented language meets demands of distributed processing, Electronis, Nov.30, 1982

[7] J.L.Petersen: Petri Nets, ACM Comp.Surveys 9 (1977)

[8] W.M.Dehnert, V.Haase: High level language structures for distributed real-time programs. SOCOCO 79, Prague (1979)

[9] P.Brinch Hansen: The Architecture of Concurrrent Programs. Englewood Cliffs (1977)

[10] P.Brinch Hansen: Distributed Process - a Concurrent Programming Concept. CACM 21/11, p.934-941 (1978)

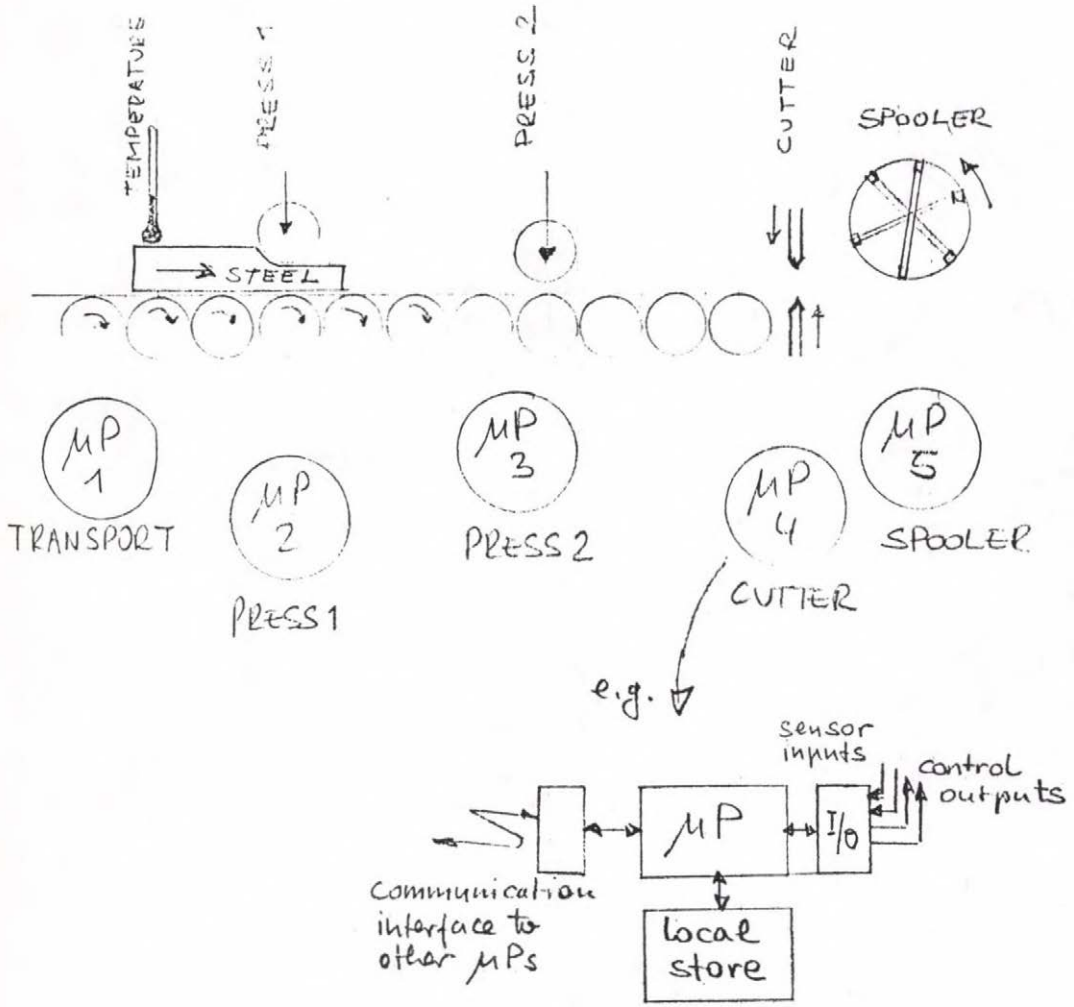


Fig. 1 : Multi-Microprocessor System in Process Control
(steel rolling mill)

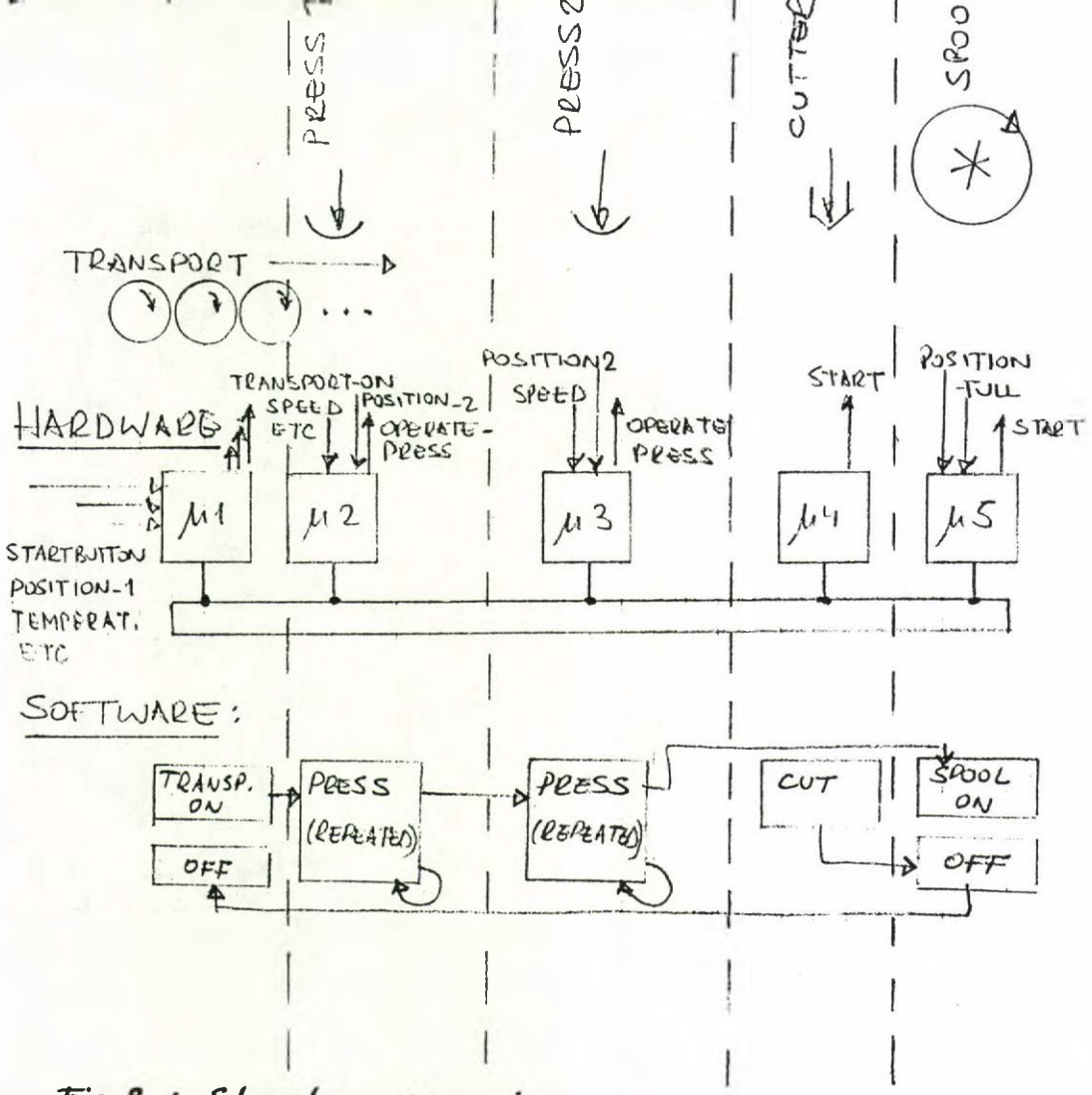


Fig. 2 : Structure mapping

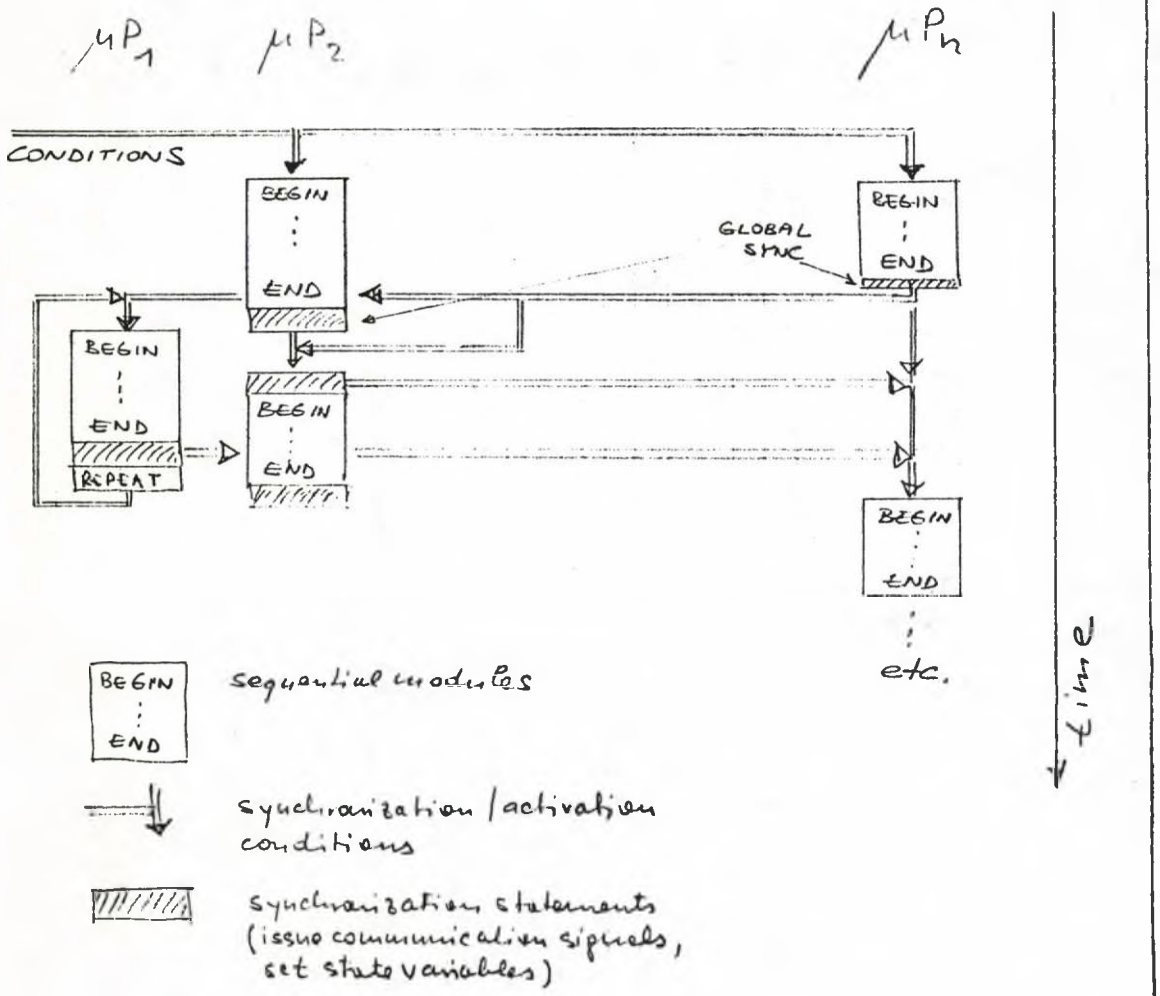
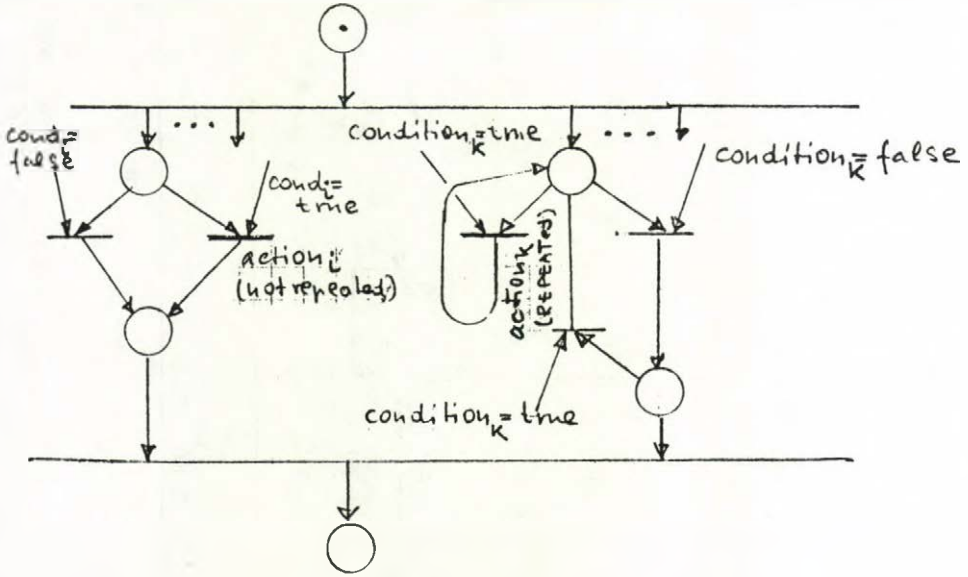


Fig.3 : Program construction out of modules [example]



left hand side : non repeated ("IF") conditional actions

right hand side : repeated ("DO") conditional actions

Fig. 4 : Parallel Constructs
[Petri Net equivalent]

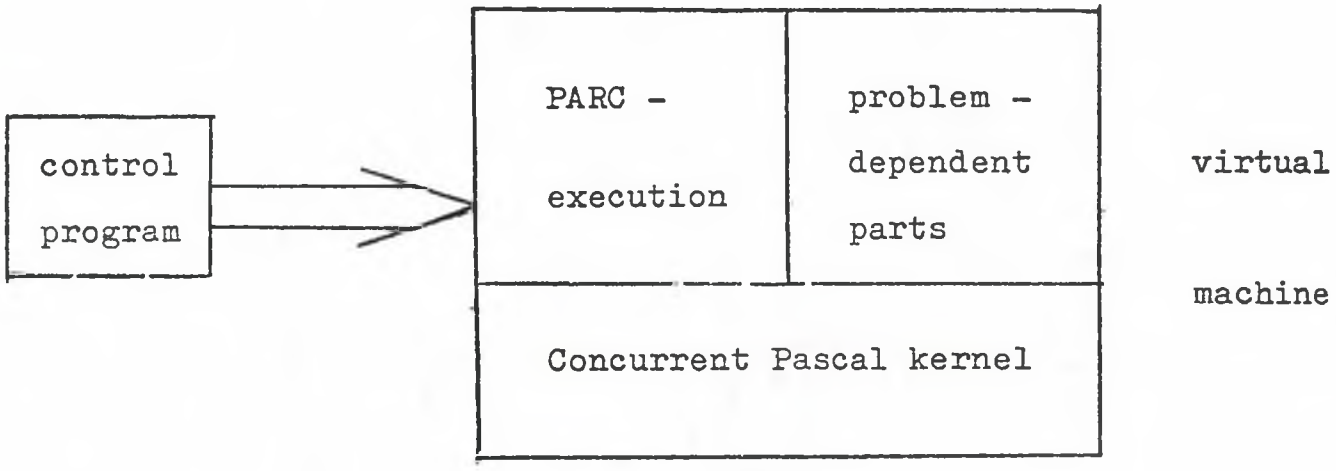
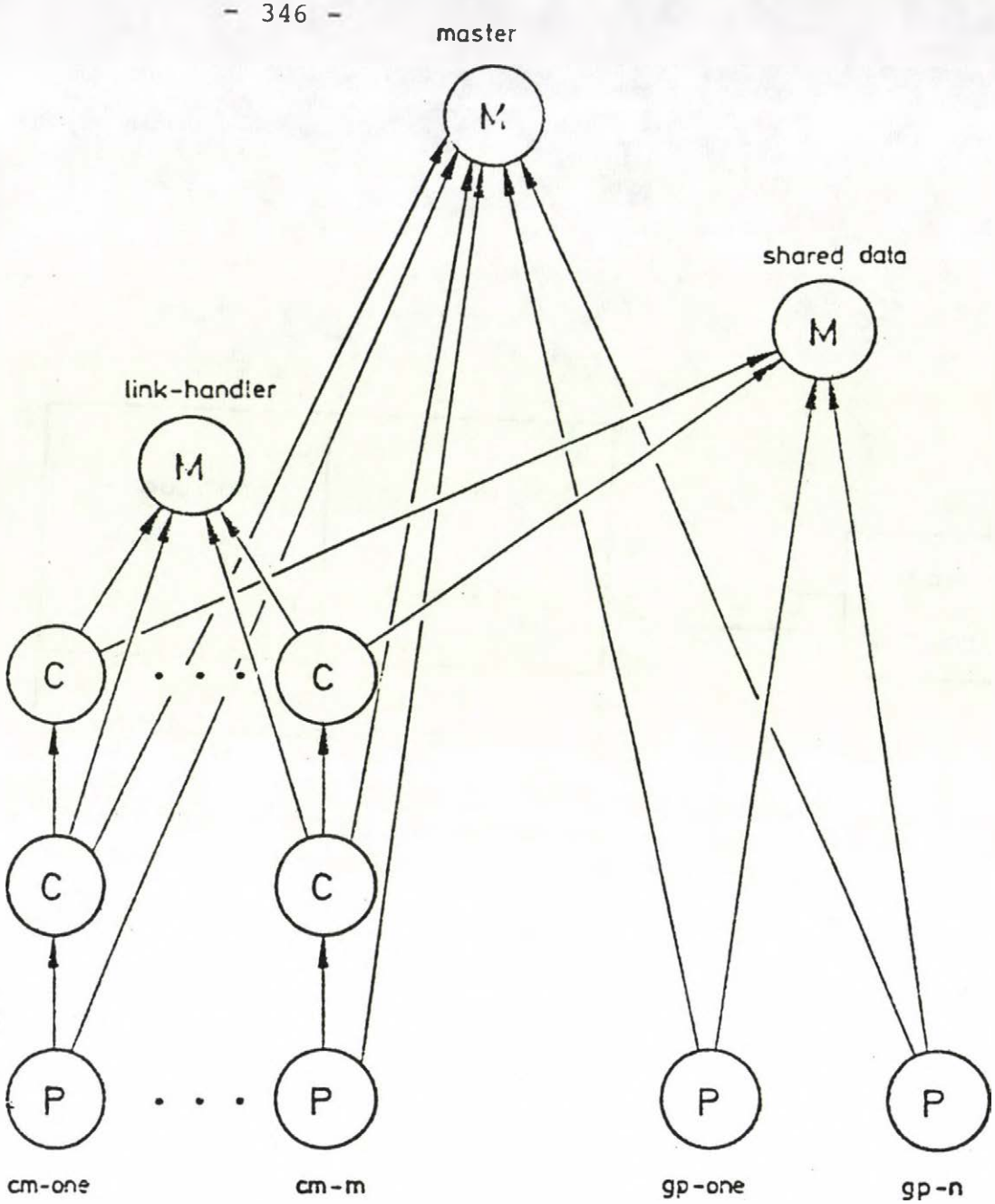


Fig. 5

TRANSLATION OF THE LANGUAGE STRUCTURES



KEY : cm control module
gp guarded process
P, M, C system components of type process, monitor and class

Fig. 6

ACCESS GRAPH

DESCRIPTION OF DECISION TABLES BY PSL/PSA

Svatava Machová - Bohumil Miniberger
Charles University Computing Centre
Malostranské nám. 25
118 00 Prague
Czechoslovakia

1

It is well known that decision tables can be usefully employed to show the solution to any logical problem where decision making is involved. Each decision table simply lists in tabular format all the relevant conditions of the problem together with all the possible actions, and indicates the actions to be carried out when certain conditions are true or false /or immaterial/ /ICL General Manual, 1972/.

A decision table is an important structured tool of analysis. "When applied - and applied properly - a decision table can be unmatched for clarity and precision. In addition to being a descriptive tool, a decision table can help you to think out policy in the making, to evaluate it for completeness and consistency" /DeMarco, 1979/.

At first sight, the PSL/PSA user might feel that there

is no need having any further tools to express problem specifications. But, how should we fill the gap between the user and the analyst and detect the inaccuracies in the word specification of the problem? Let us continue the quotation from T. DeMarco so as to show when a decision table should be used. "Suppose you query your user about his policy for charging charter flight customers for certain inflight services, and he tells you something like this: If the flight is more than half-full and costs more than $ 350 per seat, we serve free coctails unless it is a domestic flight. We charge for cocktails on all domestic flights ... that is, for all the ones where we serve cocktails". Expressing the policy in the form of a decision table solves all of these problems.

		RULES							
		1	2	3	4	5	6	7	8
CONDITIONS	1. Domestic	Y	N	Y	N	Y	N	Y	N
	2. Over half-full	Y	Y	N	N	Y	Y	N	N
	3. Over $ 350	Y	Y	Y	Y	N	N	N	N
ACTIONS	1. Coctail served	X	X	-	?	X	?	-	?
	2. Free	-	X			-			

The condition entries are 'Y' for YES, 'N' for NO. The action entries are 'X' for execute action, '-' for do not execute action.

Interrogation marks and gaps must not occur in the decision table. This is because they indicate that the decision process is described in an incomplete way and all that a system analyst can do is to make the user increase the precision of the description of his decision system using further queries. Hence, the use of decision tables is especially convenient for the description of decision process in all cases in which the subpolicy selection depends upon combinations and conditions. This is an appropriate tool for an interview carried out by the analyst, who is almost ignorant about the real system in question (he is an analyst by profession).

The decision tables can also be used in describing programme specification - another form of flowchart record.

2

Because the PSL/PSA makes it possible to describe on different levels of analysis very adequately both the information structures and the data structures, it appears convenient to use it for recording the procedural aspect of the description phenomenon by means of decision tables on the two levels.

There are different processors, which, on the basis of decision tables, design programmes for decision

processes /Pollack, 1965/. To our knowledge, it was U. Liebe /Liebe, 1981/ that proposed the outputs of a certain processor processing decision tables and designing a certain programme based on them to be stored as a comment PROCEDURE in PSL/PSA. Contrary to this, it is our concern to make it possible for the user to describe the decision tables by means of PSL/PSA even before the decision table is processed by a special processor as a programme in some programming language. To enable efficient operation with decision tables in the frame of PSL/PSA, it is necessary to describe them not by comments, but through PSL statements.

3

It is evident that out of the types of objects existing in the PSL/PSA A 5.1 version /Language Reference Manual, 1981/ the objects of the type CONDITION, EVENT and PROCESS appear to be semantically suitable for a description of decision tables. The conditions of the decision table can be described as a PSL object of the type CONDITION. The sum of all the states of conditions from the decision table important for evoking certain activities can be described as a PSL object of the type EVENT. The activities can be described as PSL objects of the type PROCESS.

In the A 5.1 version, the following relationships between the object of the type CONDITION and that of EVENT are admissible:

CONDITION user-name,

BECOMING $\begin{Bmatrix} \text{true} \\ \\ \text{false} \end{Bmatrix}$ $\begin{Bmatrix} \text{CALLS} \\ \\ \text{CAUSES} \end{Bmatrix}$ EVENT-name/s/,

If a name of a new relationship, e.g. 'CO-CAUSES' is inserted into the empty braces between CALLS and CAUSES, the only PSL modification /together with the complementary 'CO-CAUSED BY' relationship/ will be materialized that is, to our opinion, necessary for the user to describe the decision tables. To facilitate the creation of a new output reports representing decision tables we consider it convenient to ask the user to have all the EVENT's of one decision table necessarily connected by the KEYWORD of the same name.

Shouldthe user have the chance to operate with ISLDS, /Teichroew-Macasovic-Hershey-Yamamoto, 1979/, and should he be interested in a decision table description, he can use this proposal for his inspiration.

Based on the description of decision table proposed by means of PSL/PSA, A 5.1 version, output reports of various kinds can be effected approaching at maximum to the usual system of decision table representation. An output report of this kind could be of the following form:

DECISION TABLES

KEYWORD DT-1

	EVENTS				
	E-1	E-2	E-3	E-4	E-5
CONDITION-1	Y	Y	Y	N	N
CONDITION-2	Y	Y	N	Y	N
CONDITION-3	Y	N	-	-	-
PROCESS-1	X	-	X	-	-
PROCESS-2	-	X	X	-	X
PROCESS-3	-	-	-	X	-
PROCESS-4	-	X	-	X	-

CONDITION-1 - condition-name

CONDITION-2 - condition-name

CONDITION-3 - condition-name

PROCESS-1 - process-name

PROCESS-2 - process-name

PROCESS-3 - process-name

PROCESS-4 - process-name

At the Charles University Computing Centre Prague, the PSL/PSA A 5.1 version has not been available up to now, therefore the above-mentioned proposal has not been

implemented as yet. The decision tables had to be described in the A 2.1 version, which is much less elegant. A detailed description of the solution is not interesting for an actual user of higher PSL/PSA versions. It should only be added that the set of all the states of conditions of the decision table relevant for evoking certain activities have been described by us as objects of the type EVENT, but individual conditions have been described as ATTRIBUTES of that EVENT. ATTRIBUTES acquiere values identical with the states of conditions.

4

If it is possible to generate, in an automatized way, the DATA DIVISION report for a programme written in the language Cobol from the description of data structures /Chikofsky-So-Gunnarson, 1980/, and if it is known that the Cobol preprocessors convert decision tables directly to the Cobol source programmes, then it is justified to assume that it would be possible by means of the decision table representation in PSL/PSA to rationalize appreciably the writing of Cobol programmes. In the Appendix to this paper /1/, the DT-1 given above is described by means of PSL statements /and the relation 'CO-CAUSES' between the objects CONDITION and EVENT proposed above is used in it/, /2/ the corresponding DT is given in its classic form,

/3/ a part of the programme in language like Cobol which, should be generated on the basis of the DT-1 desription is given.

2 DECISION TABLE DT-1

	RULES				
	1	2	3	4	5
1. PODMINKA-1	Y	Y	Y	N	N
2. PODMINKA-2	Y	Y	N	Y	N
3. PODMINKA-3	Y	N	-	-	-
1. P-1	X	-	X	-	-
2. P-2	-	X	X	-	X
3. P-3	-	-	-	X	-
4. P-4	-	X	-	X	-

3 THE TOPICS OF THE PROGRAMME IN LANGUAGE LIKE COBOL WHICH SHOULD BE GENERATED FROM THE DESCRIPTION OF THE DT-1 IN PSL/PSA

```
DECISION-TABLE-DT-1.
    IF   PODMINKA-1
         IF   PODMINKA-2
              IF   PODMINKA-3
                sequence of statements for P-1
              ELSE sequence of statement for P-2 and P-4
           ELSE sequence of statement for P-1 and P-2
              ELSE IF PODMINKA-2
                   sequence of statements P-3 and P-4
                   ELSE sequence of statements P-2.
```

```
DEF     EVENT          E-2;
        TRIGGERS         P-2,
                         P-4;
        KEY      DT-1;

DEF     EVENT          E-3;
        TRIGGERS         P-1,
                         P-2;
        KEY      DT-1;

DEF     EVENT          E-4;
        TRIGGERS         P-3,
                         P-4;
        KEY      DT-1;

DEF     EVENT          E-5;
        TRIGGERS       P-3;
        KEY      DT-1;

DEF     PROCESS        P-1, P-2, P-3, P-4;
        KEY      DT-1;
```

APPENDIX

1 DESCRIPTION OF THE DT-1 BY MEANS OF PSL STATEMENTS

```
DEF     CONDITION       PODMINKA-1;
        BECOMING TRUE     CO-CAUSES       E-1,
                                          E-2,
                                          E-3;
        BECOMING FALSE    CO-CAUSES       E-4,
                                          E-5;
        KEY     DT-1;

DEF     CONDITION       PODMINKA-2;
        BECOMING TRUE     CO-CAUSES       E-1,
                                          E-2,
                                          E-4;
        BECOMING FALSE    CO-CAUSES       E-3,
                                          E-5;
        KEY     DT-1;

DEF     CONDITION       PODMINKA-3;
        BECOMING TRUE     CO-CAUSES       E-1;
        BECOMING FALSE    CO-CAUSES       E-2;
        KEY     DT-1;

DEF     EVENT           E-1;
        TRIGGERS          P-1;
        KEY     DT-1;
```

LITERATURE:

CHIKOFSKY E. - SO L.K. - GUNNARSON K., Data Division /DDIV/ report in PSA Version A 5.2. ISDOS Technical Memorandum 326, September 23, 1980.

DeMARCO T., Structured analysis and system specification. New Jersey, 1979.

FULTON J., Some preliminary suggestions for enhancements to PSA capabilities. ISDOS Newsletter 13, 1981, No. 4, Enclosure B.3.

ICL GENERAL MANUAL 4139. Appendix F. ICL Beaumont, Old Windsor, Berks., 1972.

LIEBE U., PSL/PSA supported microcode development. ISDOS Newsletter 13, 1981, No. 4, Enclosure A.3.

LANGUAGE REFERENCE MANUAL, Problem Statement Language, ISDOS, Ref. 81252-0351, Ann Arbor, 1981.

TEICHROEW D.- MACASOVIC P. - HERSHEY III.E.A. - YAMAMOTO Y., Application of the entity-relationship approach to information processing systems modeling. /In/ International Conference or Entity-Relationship Approach to Systems Analysis and Design. Los Angeles, 1979, 23-51.

PSL/PSA - A METHODOLOGICAL TOOL FOR THESAURUS CREATION

Svatava Machová
Charles University Computing Centre
Malostranské nám. 25
118 00 Prague, Czechoslovakia

1

In general, the PSL/PSA A5.1 version may be regarded as a tool for describing a real system, which makes it possible to distinguish a certain number of types of objects within this real system. Each object may acquire certain properties /determined by the PSL/PSA designers/ and can enter into certain interrelationships /also determined by the PSL/PSA designers/. The PSL/PSA designers gave names to the admissible types of objects, properties and relationships /e.g. SET, ATTRIBUTE, DERIVES/. By the choice of names, however, they largely determined the meanings that the users may assign to the types of objects, properties and relationships. Although the linguists gave evidence about the fact that the words occurring outside any context are of no meaning /Hjelmslev, 1953/ and that they acquire a meaning only within a concrete text, a current user is not aware of this fact and expects each separate word outside the context to be of some meaning of its own. He assumes

wrongly that it is the meaning that he assigns to the particular word most frequently within his own given sphere.

We feel that it is because the individual users assigned one single meaning to each of the words used in PSL/PSA that some kinds of PSL/PSA modifications leading eventually to the creation of the ISLDS /Bodart - Teichroew, 1981/ were required.

We will demonstrate in one instance that even PSL/PSA A5.1 version can be operated as a metasystem provided that there is a general underlying linguistic hypothesis saying that isolated words are semantically empty and that an arbitrary meaning based on convention between the users can hence be assigned to them.

2

For information storage and retrieval systems thesauri have to be designed. Roughly speaking, thesaurus is a semantic dictionary consisting of descriptors and structural interrelations between them. Usually the following relations are distinguished between descriptors: equivalence, generic hierarchy, partivity, association and antonymy. Thesauri are usually large and in the course of their creation it is convenient to be assisted by a computer.

The fact that thesauri are usually not created with the assistance of a computer is due to the circumstances

that (i) designing a software for thesaurus creation is demanding and costly and (ii) a designer of thesaurus with little experience does not know how to specify the requirements for the software needed.

Although it is obvious that a software designed exclusively for thesaurus creation offers an ideal solution, there exist other possibilities for computer-assisted thesaurus creation. It is posiible, indeed, to make use of a software readily designed for other purposes. Before starting the search for it, it is necessary to give an answer to the question as to what are the properties that a software for thesaurus creation is supposed to possess.

I believe that the following properties are essential:

- ability to record different types of relations between descriptors,
- ability to create automatically a complementary relation to each relation described,
- easy updating of information stored,
- possibility of presentation of the stored data by means of output reports enabling a differential view of the same data,
- interactive processing,
- easy implementation on the computer available.

With view of the above properties I consider most suitable generally for the creation of thesauri the data dictionaries and that part of the software for automated

system modeling which fulfills the function of data dictionaries without being so labelled.

3

The Charles University Computing Centre Prague began to work in designing software for a thesaurus of terms for banking regulations created by the Czechoslovak State Bank /Hudec-Machová, 1983/. PSL/PSA proved to be a very appropriate methodical tool of thesaurus creation, provided we accept the convention that some PSL reserved words will be assigned specific meanings. The following specific meanings of PSL reserved words were adopted for the thesaurus of terms for banking regulations:

PSL Reserved Words	Meaning Assigned
SET	thesaurus,
ENTITY	descriptor /one-word or multi-word/,
DESCRIPTION	definiton/s/ of the descriptor,
SOURCE	publication/s/ where the descriptor has been defined and/or occured,
SYNONYM/S/	descriptors in equivalence relation /so-called 'alias' relation/,
RELATION	relations of generic hierarchy, partitivity, association and antonymy between the descriptors,

ATTRIBUTTES	The class of objects of this type appears to be ideal for this particular application. It makes it possible to assign arbitrary properties to an object, to name them and to assess the value that the property acquires in connection with the object in question. For the descriptors /i.e. for the objects of the type ENTITY/ the following properties have been chosen: the date of insertion into the database, the occurrence of the descriptor in some publication /up to 10 potential occurrences/, foreign-language equivalents /English, French, German, Russian/,
KEYWORDS	connecting link of objects of different types having relation to one SOURCE, or descriptors related to other descriptors by the same type of relation,
PROBLEM DEFINER	the same as in current use,
MAILBOX	the same as in current use,
CONSISTS OF	the same as in current use,
CONTAINED IN	the same as in current use.

On the adoption of this convention, as far as the meaning of the PSL reserved words is concerned, the thesaurus was materialized in the form of a semantic network. The nodes of the network are constituted by objects of several kinds:

(i) descriptors, (ii) properties of descriptors (creating terminal nodes), (iii) sources in which the descriptors occur, (iv) relations stated between the descriptors. The edges between the objects remain labelled as proveded for by PSL.

At present, works have been finished on the experimental sample of thesaurus containing 110 descriptors. The thesaurus is assumed to contain 3,000 descriptors the materials for book edition of the thesaurus being made up by the PSL/PSA reports.

Even in this relatively small extent of thesaurus the multiple advantage of the software chosen became manifest:

- easy modification of the stored data /mocifications are rather frequent at the early stages of thesaurus creation/,
- easy description of semantic relations between descriptors /as can be seen from the Appendix, these relations are fairly rich/,
- easy description of the place of occurrence of the descriptor in the publications concerned and of the degree of importance of the occurrence,
- easy description of publications,
- easy registration of workers responsible for the retrieval in the publications,
- easy supply of information both to the designers and the users of the thesauri. The following PSL/PSA output reports are convenient: NG with various parameters, FPS with

various parameters, DICTIONARY with various parameters, KWIC,

- possibility to supply information as reply /in the form of a report/ to query /in the form of PSA commands with parameters/ in such a way that the reply structure should be in keeping with the functional perspective of the query.

The software chosen fails to enable to materilize a facet approach to the description of generic relations between descriptors, but in the subject area in question it is possible to do without facets.

The way of operation with a thesaurus created by means of PSL/PSA software is comfortable and agreeable. It can be assumed that it will contribute to frequent utilization of thesaurus by managing workers and thus become one of the basic resources of managing information.

LITERATURE:

AITCHINSON J. - GILCHRIST A., Thesaurus construction. London, 1972.

BODART P. - TEICHROEW D., Les outils d'aide à la conception d'un système d'information. Informatique et Gestion, 1981, No. 125, 47-55.

HJELMSLEV L., Prolegomena to the theory of language. London, 1953.

HUDEC C. - MACHOVÁ S., Automation of control information in a bank. Paper to be read at the INFOSEM'83, Piestany, Slovakia, May 1983.

PSA VERSION [illegible] CHARLES UNIVERSITY/VCUK APR 21, 1983 11

FORMATTED PROBLEM STATEMENT

PARAMETERS FOR FPS

```
    NAME=OBRAT NOINDEX PRINT NOPUNCH SMARG=5 NMARG=20 AMARG=10 BMARG=25 RNMARG=7
    DESG ONE-PER-LINE DEFINE COMMENT NONEW-PAGE NONEW-LINE

    1 ENTITY                                        OBRAT;
    2       DESCRIPTION;
    3               PLNE ZNENI HESLOVEHO SLOVA:
    4               'OBRAT',
    5               DEFINICE:
    6               OBRAT PENEZNI JE UHRN PLATEB V NARODNIM HOSPODARSTVI ZA
    7               URCITY CASOVY USEK; TYTO PLATBY JSOU VZDY SPOJENY JEDNAK S
    8               REALIZACI CEN PRODAVANEHO ZBOZI A SLUZEB, JEDNAK S
    9               VYROVNAVANIM SPLATNYCH PLATEB RUZNEHO DRUHU,
   10                                                                   (FS-73)
   11               OBRATEM SE ROZUMI UHRN PRIJMOVYCH A VYDAJOVYCH POLOZEK NA
   12               JISTEM UCTU                                         (SSJC);
   13       KEYWORDS        ZDROJ-A158;
   14       ATTRIBUTES ARE
   15            DATUM-ZARAZENI-DO-TEZAURU
   16                            D'1982-05-19',
   17            EKVIVALENT-ANGLICKY
   18                            TURNOVER,
   19            EKVIVALENT-FRANCOUZSKY
   20                            CHIFFRE-D-AFFAIRES,
   21            EKVIVALENT-NEMECKY
   22                            GELDUMSATZ,
   23            EKVIVALENT-RUSKY
   24                            DENEZNYJ-OBOROT;
   25       CONTAINED IN    TEZAURUS-VYRAZU-BANK-PREDPISU;
   26       /* RIGHT */ RELATED TO
   27            BANKOVNI-UCET
   28             VIA            ASOCIACE-7;
   29       SOURCE IS       A158,
   30                       FINANCNI-SLOVNIK,
   31                       SPISOVNY-SLOVNIK-JAZ-CESKEHO;
   32
   33 EOF EOF EOF EOF EOF
```

ENTER COMMAND (AND ANY PARAMETERS)

FORMATTED PROBLEM STATEMENT

PARAMETERS FOR FPS

```
NAME=UVER NOINDEX PRINT NOPUNCH SMARG=5 NMARG=20 AMARG=10 BMARG=25 RNMARG=70
DESG ONE-PER-LINE DEFINE COMMENT NONEW-PAGE NONEW-LINE

ENTITY                                      UVER;
    DESCRIPTION;
        PLNE ZNENI HESLOVEHO SLOVA:
        'UVER'.
        DEFINICE:
        UVER JE:(1) VZTAH VZNIKAJICI PRI PREDANI HODNOTY DRUHYM K
        DOCASNEMU POUZITI; (2) FORMA NAVRATNE REDISTRIBUCE DOCASNE
        NEBO TRVALE UVOLNENYCH ZBOZNICH A PENEZNICH FONDU.
                                                              (FS-73)
        UVEREM SE ROZUMI POSKYTOVANI PENEZ NEBO ZBOZI S TIM, ZE
        BUDOU VRACENY NEBO ZAPLACENY POZDEJI.                  (SSJC);
    KEYWORDS           ZDROJ-1158;
    ATTRIBUTES ARE
        DATUM-ZARAZENI-DO-TEZAURU
                          D-1982-05-19,
        EKVIVALENT-ANGLICKY
                          CREDIT,
        EKVIVALENT-FRANCOUZSKY
                          CREDIT,
        EKVIVALENT-NEMECKY
                          KREDIT,
        EKVIVALENT-RUSKY
                          KREDIT;
    CONTAINED IN    TEZAURUS-VYRAZU-BANK-PREDPISU;
    /* LEFT */ RELATED TO
        VKLAD VIA         ANTONYMIE-1;
    /* LEFT */ RELATED TO
        PROVOZNI-UVER
         VIA              RODOVA-PODRAZENOST-13;
    /* LEFT */ RELATED TO
        INVESTICNI-UVER
         VIA              RODOVA-PODRAZENOST-14;
    /* LEFT */ RELATED TO
        DEVIZOVY-UVER
         VIA              RODOVA-PODRAZENOST-15;
    /* RIGHT */ RELATED TO
        UVEROVY-VZTAH
         VIA              RODOVA-PODRAZENOST-11;
    SOURCE IS         A453,
                      FINANCNI-SLOVNIK,
                      SPISOVNY-SLOVNIK-JAZ-CESKEHO;

EOF EOF EOF EOF EOF

NTER COMMAND (AND ANY PARAMETERS)
```