THE SOFTWARE KNOWLEDGE BASE

Bertrand Meyer

Computer Science Department, University of California
Santa Barbara, California 93106 (USA)

(On leave from Electricité de France, Clamart (France))

ABSTRACT

We describe a system for maintaining useful information about a software project. The "software knowledge base" keeps track of software components and their properties; these properties are described through *binary relations* and the *constraints* that these relations must satisfy. The relations and constraints are entirely user-definable, although a set of predefined *libraries* of relations with associated constraints is provided for some of the most important aspects of software development (specification, design, implementation, testing, project management).

The use of the binary relational model for describing the properties of software is backed by a theoretical study of the relations and constraints which play an important role in software development.

Keywords: Software engineering tools, configuration management, project management, formal description of software engineering concepts.

1. INTRODUCTION

Studies have repeatedly shown that management problems are one of the primary sources of delays and failures in large software projects (see e.g.[5]).

If bad management is due to bad managers, one can hardly expect that advances in software engineering will alleviate the problem. But bad management, or rather bad organization, often has another cause: the sheer difficulty of mastering the various aspects of a project, and in particular of controlling change. Project managers and project members alike have trouble keeping track of what is going on. As the project develops, its "entropy" increases and it becomes increasingly difficult to maintain a clear picture of the state of its various components. Here good tools can play a major role.

The effort reported in this paper aims at providing a unified base of supporting tools for various aspects of software development. To this end, we introduce the notion of a **software knowledge base**, that is to say a repository of all useful project information.

The software knowledge base is used by managers and programmers to keep track of all interesting properties of the software components and their relationships. The software *components*, as defined here, include all the relevant project elements: program modules, data definitions, requirements, user manuals and other documentation, specifications, design documents, test data, test results, schedules, tasks, personnel data, budgets etc. The *relations* between these components may be of diverse kinds: we may want to record the fact that a certain module of the design implements a certain module of the specification, that a certain program module uses a certain data definition module, that a certain task is assigned to a certain person, etc.

We use the expression "software knowledge base", or SKB, to denote the compendium of information associated with a software project. The system used to record, access and manipulate this information, as described in this paper, is called the SKB *system* whenever there might be a confusion.

Several aspects of the SKB system are present in previous project management systems. The ISDOS system[34] is a set of project documentation tools, which make it possible to record project information as relations between entities of various predefined kinds; these ideas were further developed in the SREM system[3] written at TRW, which particularly emphasized the notion of traceability (i.e. ability to locate over the whole data base the consequences of a change made to some element). Simpler yet very useful tools for **configuration management** and **version control** are gaining acceptance: Make[15] and SCCS[32] on Unix, DEC's CMS, Softool's CCC, the "System Version Control" component of Gandalf[20], Adèle[14], RCS, etc. The idea of collecting all useful project documents in a single database is expounded in the Stoneman report[10], and used in a current TRW development, the "Software Master Database"[6]. The use of relations in software environments is advocated in[11], which relies on general (n-ary) relational databases;[26] shows that binary relations may be applied to various aspects of programming. The SKB project builds on all these ideas but emphasizes some original, and in our view essential, design criteria, which we shall now describe.

2. DESIGN CRITERIA

2.1. Simplicity

The SKB system should be easy to learn and use. Managers and programmers have enough to do already; they should not be required to go through an extensive training period before they can effectively use the SKB system.

A necessary condition for ease of learning and use is to base the whole system on a simple and uniform conceptual framework.

2.2. Method-, language- and system- independence

The SKB system is a set of tools, not an integrated methodology. Although its consistent use naturally leads to some sound methodological practices, it should be viewed as a way of helping project managers and developers, not as a disruption of current development practices.

Thus the SKB system should not conceptually imply the use of any particular methodology, programming language, computer or operating system. It should blend well with other software engineering tools.

We will refer to this criterion as the "independence" criterion (as a shorthand for method-, language- and system-independence).

2.3. Adaptability

Not only should the system be compatible with existing methods or languages: it should be able to provide efficient support for specific methods or languages in use in, say, a company.

Thus the natural counterpart of independence is the ability to parameterize.

2.4. Whole life-cycle coverage

The SKB system should provide benefits across the entire life-cycle of a software project. Although this criterion may restrict the power of SKB tools as applied to a specific life-cycle stage, it is essential in view of the fact that non-trivial projects usually have a long history. A system that would only apply to, say, the initial phases of specification and design, would stand little chance of playing a significant role: so much of the software process is evolution, refinement and extension of systems occurring after the first "cycle" has been completed.

2.5. Support for system semantics

Many of the systems quoted in section 1 have little, if any, notion of what the objects being manipulated really "are". Most configuration management systems, for example, focus on just one attribute of objects, their time stamp, and know of just one relation, the dependency relation (there is usually also the notion of a "permission" attribute in the systems which support protection). These systems are not equipped to deal with other properties of the objects such as the *"A is an implementation of B"* relation quoted above.

On the other hand, some of the more complex systems do know about "types" of objects (e.g. specification, test data set, etc.), but then they violate the independence criterion since these types are defined once and for all. The problem is thus to be able to record semantic properties of software objects while retaining flexibility.

2.6. Formal analysis

The design of the SKB system was based on a systematic analysis of the properties of software project elements; some elements of this analysis are given below (section 5), in the form of a review of software relations and their abstract properties (constraints).

This approach contrasts with most published work on software engineering tools: although the necessity for a systematic requirements analysis is one of the tenets of software engineering, it seems to have seldom been applied, let alone in a formal way, to software engineering tools and environments. The formal specifications we know in this field are *a posteriori* exercises applied to existing designs, e.g.[18], which describes some aspects of ISDOS, and[12] which describes the system version control component of Gandalf. The analysis outlined in section 5 is not a complete specification of the SKB system, but provides a sound (we hope) theoretical basis for the system.

2.7. Object-independence

A software knowledge base is a **model** of a certain set of software objects and their relations. The model is conceptually and physically distinct from the objects themselves; this is in contrast with systems that essentially add project and configuration management information to the object representations (usually files on a conventional host system). In our approach, the SKB is a separate entity; objects are modeled by SKB elements, called "atoms" below.

Thus a reference to the object modeled by an SKB atom (e.g. the file containing a program or other software object) will merely be considered as one of the *attributes* of the atom (the notion of attribute is made more precise below).

Such an approach has advantages and drawbacks. The advantages are simplicity and portability; the SKB system can be built on top of any operating system without undue modification to this operating system. The approach also makes it possible to keep the model (the SKB) on one computer and the objects themselves on another if it is deemed preferable to separate the development machine from the management machine.

On the other hand, the approach taken makes it impossible to ensure consistency: one cannot prevent users from modifying the objects without making the corresponding changes in the model. However, regardless of the decision taken, it is hard to ensure consistency anyhow unless one is to build a management system that replicates most of the functions of an operating system. For example, if one wants to guarantee that the management system knows about all changes brought to the objects, then the management system should include such utilities as text editors and the like. We did not want to follow this path.

Thus we prefer to stick to the more modest goal of providing a set of management tools on top of an existing operating system, with an open architecture which makes it possible to combine these tools with other software tools. It is the responsibility of the project members to maintain an accurate SKB about the project. In other words, we accept the possibility that the SKB system may be fooled, as a price to pay for the simplicity, flexibility and independence (in the above sense) of that system.

Obviously, efforts should be made to improve the consistency of the SKBs. In particular, specific interfaces may be built between the SKB system and the host system so that information may be entered automatically into the SKB, as a result of operations performed in the host system (e.g. a compilation or an editing session).

Our approach thus follows the example set by the Make system[15], which achieves simplicity by relying on dependency information provided explicitly by programmers; this system having proved to be useful, efficient and easy to use, other researchers have been able to come up with tools[35] that automatically feed dependency information into Make for specific cases (source programs in C, Pascal, Fortran, Lex and Yacc in the reference cited).

3. THEORETICAL BASIS

The notion of software knowledge base is based on a small number of concepts: atoms, attributes, relations, constraints and actions.

3.1. Atoms

The objects in the knowledge base, associated with physical objects of the software project, are called atoms. As implied by the "object-independence" criterion discussed above, the atoms have no immediate connection with the objects they represent; they are meaningful for the SKB operations only, and their properties are only defined through their attributes, relations with other atoms, and constraints on these relations.

3.2. Attributes

Atoms may have attributes. Attributes are user-definable, although some predefined attributes are available. The value of an attribute may only belong to one of a small number of predefined types such as *Integer, String, Time, File.* The values of the last type are are references to files supported by the operating system (in a non-standard system that does not have files, we may have to replace this by a more general notion of "object").

Typical predefined atom attributes are *time_of_last_change*, yielding values of type *Time*; *atom_type*, yielding values of type *String* (some possible types for atoms are predefined, e.g. "procedure", "requirement", "test data", etc., but new ones may freely be added); and *representation*, yielding values of type *File.*

Attributes may not be of complex types; in particular, they cannot yield atoms. For anything but simple properties of atoms, relations should be used instead (see below).

3.3. Relations

The heart of an SKB consists of a series of facts about the software project, expressed as links between atoms. Each of these links expresses the fact that a certain relation holds between two atoms *a* and *b*. Examples (complementing those in the introduction) are "*a* is defined in *b*" (where *a* is a procedure and *b* a package in Ada), "*a* is a member of *b*" (where *a* is a person and *b* a project), "*a* is the formal expression of *b*" (where *a* is a module in a specification and *b* a paragraph of the requirements document). More examples will be found in section 5.

The SKB system only uses binary relations; the reason is that binary relations are mathematically simple, have nice properties, and provide an intuitively appealing way to describe structural properties of systems. From the theoretical standpoint, any system that can be described using general relations (as e.g. with a relational data base management system) can be described with binary relations[8], and algorithms have been proposed to efficiently translate a binary schema into a more efficient *n*-ary one[31].

In practice, we have indeed found binary relations to be adequate for modeling properties of software objects. This is illustrated by the analysis in section 5 - where it will be seen that we did find one case where a ternary relation seems necessary.

3.4. Constraints

Attributes and relations constitute by themselves an empty shell; they describe the structural connections between software objects (the "syntax" of the project), but not their deeper properties (the "semantics"). The latter may be expressed by defining *constraints*, or conditions on the relations and attributes, which must be satisfied for the SKB to be in a consistent state. A simple and important example of constraint is the "dependency" constraint maintained by tools such as Make, which expresses that the value of the *time_of_last_change* attribute should be greater (i.e. more recent) for every atom than for every atom to which it is connected by any relation that may be characterized as a "dependency" relation. But many other useful constraints may be defined on software systems; some will be given below.

Constraints will be expressed as mathematical relational predicates involving relations, attributes and atoms. The abstract formalism used to construct and manipulate a software knowledge base is called the Calculus of Relations, Attributes and Constraints (CRAC).

3.5. Actions

An action is associated with a constraint and specifies steps to be taken when the constraint is violated by certain objects, following manipulations of the SKB. Actions are not strictly part of the SKB system since they may involve commands to the operating system; the SKB system provides the interface, and ways to pass attributes of the atoms (e.g. file names) to the host system.

4. STRUCTURE OF THE SYSTEM

The structure of the SKB system follows from the design criteria of section 2 and the theoretical basis described in section 3. It is represented in figure 1.

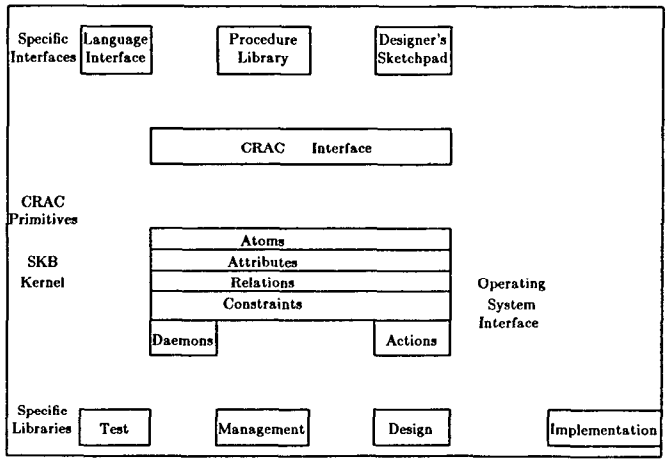

Figure 1: Structure of the SKB System

The **kernel** of the SKB system provides the basic mechanisms for creating, accessing and updating the SKB entities: atoms, attributes, relations and constraints.

In connection with constraints, we introduce the concept of **daemon**. A daemon is a mechanism associated with a constraint, which monitors the SKB in order to detect possible violations of the constraint as the information in the SKB is being updated (i.e. links between atoms are modified, new atoms are entered, attributes are changed, etc.). When it finds that such a violation has been made, the daemon will report the inconsistency and trigger the action associated with the constraint, if there is one.

Daemons raise an interesting implementation problem: in a large SKB involving many atoms, attributes and relations, it is essential to find ways to avoid searching the whole structure (mathematically, a multigraph) for the consequences of a simple change. Work on related topics has been done previously in connection with artificial intelligence [25] and interactive graphics [17].

The SKB kernel is accessible through a set of primitives, the "CRAC primitives", which implement the calculus of relations and constraints, i.e. all the useful operations on the knowledge base. These operations are made available through a uniform interface, the "CRAC interface"; the idea here is that the SKB functions (like those of any good data base management system, or more generally of any good software engineering tool) should be equally accessible to interactive users, non-interactive users, and other programs (this is an implementation of what may be called "Strachey's principle" from the quotation of Christopher Strachey in Scott's preface to [33]: "decide what you are going to say before you decide how you are going to say it").

Thus the CRAC interface does not favor any of these types of access. Several higher-level interfaces should be provided; figure 1 lists three:

- the procedure library, which makes CRAC primitives usable from programs (e.g. other software tools), written in ordinary programming languages;
- the CRAC language, which makes it possible to express CRAC manipulations in an appropriate notation;
- the **Designer's Sketchpad**, a graphical interface to the calculus, allowing for interactive description of the atoms and relations with a graphical display and a mouse. The aim here is to avoid the gap between high-level design decisions, which are often best expressed in pictures, and the rest of the software development process.

Finally, figure 2 includes a set of "CRAC Libraries", each of which provides a set of predefined relations, attributes and constraints corresponding to an important aspect of software engineering. Examples are project management (scheduling, personnel management etc.); design (a library might provide support for a specific PDL); implementation (an Ada library manager would fit here); and testing. This last point is particularly important in our view and we see test management as one of the main benefits of the SKB system: although there is an extensive literature on program testing, very little seems to have been published on the *management* of the testing process: how to keep track of test data sets for each module, record test results, etc.

5. A TAXONOMY OF SOFTWARE RELATIONS AND CONSTRAINTS

5.1. Overview

The fact that useful relations exist among components of software systems has been pointed out by many authors. For example, Parnas[30] describes the "uses" and "invokes" relations among modules; systematic methodologies for software design have introduced the "abstraction" relation between a specification (e.g. an abstract data type) and an implementation[21]; the "isa" relation[9] is used in AI systems; the development of software development environments has recently led several researchers to consider using relational databases to keep track of the relations between the various objects needed in a software project[11,24]; at the program level, control and data dependencies play in important role in studies about code generation and optimization in compilers[2], program vectorization[23,22], static analysis[16].

Despite this frequent use of relations for software-related issues, there have been very few systematic studies of these relations; most works dealing with relations just assume that they are there, and go on using them or discussing ways to compute or implement them (an exception is[26] which introduces some program-level relations and studies their properties).

The absence of a precise definition of software relations and their formal properties is regrettable, since relations are not just vague connections between objects, nor just "tables" as in simplistic presentations of the relational database theory, but useful mathematical objects with interesting properties. We feel that a systematic study of software relations is essential to advances in software configuration management. We have started such a study[28]; some elements from that study will now be reported. The aim of this section is to present some interesting relations and the associated constraints, giving support to our decision to base the design of the SKB system on binary relations.

Of course, the relations presented here are only some of the important relations that occur in software; the SKB system is an open system and the user may introduce any relations and attributes that may be needed for a particular application, together with the associated constraints. The normal way to do this is to define CRAC "libraries"; the relations and constraints presented below would normally be part of some basic, predefined libraries.

5.2. Basic Atom Types

As mentioned above, SKB atoms are not strictly typed; they simply have "atom type" as one of their attributes. Examples of atom types are "Requirement", "Specification", "Design", "Program", "Test_data", "Variable", "Statement", "Module", "Project", "Milestone", "Staff member", "Unit cost", etc. In the spirit of the theory of abstract data types, these types are only "defined" through the relations which may hold between the corresponding atoms and the associated constraints.

In the analysis that follows, we shall be talking about types of software **objects** and relations between these objects. For the SKB project, this analysis is only interesting insofar as these properties of objects can be modeled by properties of the corresponding atoms.

5.3. Relations between atoms of different types

• ***a* contains *b***

This relation holds if and only if the object represented by *b* is a constituent of *a*. Typically, *a* will be a system, described at a certain level of abstraction (specification, design, code, documentation etc.) and *b* will be a component (module, chapter etc.) of that description. We call ***part_of*** the inverse relation ***contains*** $^{-1}$.

• ***a* models *b***

This relations holds if and only if *a* includes a description of what *b* does, that is to say if *b* is one way to do what is prescribed by *a*. We call ***instances*** the inverse relation.

Examples: the user manual for a machine **models** that machine; an abstract data type description of a type **models** an implementation of that type as a class in Simula or Smalltalk, a package in Ada etc.

5.4. Relations between atoms of the same type

• ***a* complements *b***

This relation holds if and only if *a* and *b* cooperate towards the achievement of some higher aim. For example, various procedures in the implementation of the same data type (class, package) complement each other; so do various subroutines in a numerical library, or Unix programs commonly used in a "pipe" fashion, e.g. for text processing the programs ***refer, tbl, eqn, troff***.

Constraints: ***complements*** is a symmetric relation;

$$part\_of\ ;\ contains \subseteq complements$$

In this notation, the semicolon denotes the composition of relations: ***part_of ; contains*** is the relation which holds between any two elements *a* and *c* if and only if, for some *b*, *a* is **part_of** *b* and *b* **contains** *c*. Also, if *r* **and** *s* are two relations, then $r \subseteq s$ (*r* is a subset of *s*) means that any pair of elements connected by r is also connected by *s*. The appendix describes these and other notations.

• ***a* specializes *b***

This relation holds if and only if anything which is described by *a* is also described by *b* (but some things may be described by *b* which are not described by *a*). The inverse relation, ***specializes*** $^{-1}$, may be written ***generalizes***.

Examples: In other branches of science, the Linnaean classification of living beings is based upon this relation. In software, a particular elegant implementation of this relation is the prefixing mechanism of Simula and Smalltalk: if *a* is a class whose declaration is prefixed by the name of *b*, then any property which has been given in the declaration of *b* applies ipso facto to all objects of class *b*, but this does not prevent the declaration of *a* to add any further properties which may be needed; the mechanism can be iterated. A similar mechanism exists in the Z specification language[1].

Constraints: "linear" or "hierarchical" inheritance, as in Simula and Smalltalk, means that the relation is a forest. In Smalltalk, the introduction of the "metaclass" ***Class*** makes it a tree. "Multiple inheritance would mean that a dag is acceptable.

An interesting variant of this relation occurs in many practical cases; it may be written ***a specializes b except for c*** (e.g., bats have all the properties of mammals except that they can fly). This seems very useful to model many aspects of software, e.g. Fortran 77 is "upward-compatible" with Fortran 66 (except for a few "minor" details), version 4.2 of the XXX operating system is almost compatible with version, say, 7, etc. This relation is also important in connection with modular, reusable system specifications[29]. It is a ternary relation.

• ***a* refers_to *b***

This relation holds if and only if *a* refers to *b* by its name. It can happen in a variety of ways: *a* and *b* can be objects of the same type (i.e. procedures, where ***a* calls *b***) but this is not necessary. In programming languages, a module can **refer_to** objects belonging to other modules (e.g. variables, etc.) either through the mechanism of block structure or sharing of data, or by special facilities which enable a module to "peep" into the names of entities belonging to another (**inspect** in Simula, **use** in Ada). We call ***is_referred_by*** the inverse relation.

• ***a* needs *b***

This relation holds if and only if *a* cannot be understood (or, if a program element, executed) without *b*.

Constraint: we venture the following rule:

$$needs \subseteq is\_referred\_by^{*}\ ;\ refers\_to^{+}$$

meaning that *a* possibly **needs** *b* if and only if some module *c* (which could be *a* itself) refers to both *a* and *b* directly or indirectly (the asterisk and plus sign denote transitive closures; see the appendix).

• ***a* declared_in *b***

This relation holds in block-structured languages iff *a* is declared inside *b*.

Constraint: $declared\_in \subseteq part\_of$

• ***a* shares_information_with *b***

This symmetric relation holds if and only if *a* and *b* may access some common information. In block-structured languages such as Algol 60 and Pascal, this is done through the block structure mechanism, as defined by the following constraint (valid for these languages):

$$shares\_information\_with \subseteq declared\_in^{*}\ ;\ has\_declaration^{*}$$

where ***has_declaration*** is the inverse of ***declared_in***.

5.5. Relations between program modules

The following relations apply to modules of programs (procedures, classes, packages etc.).

• ***a* calls *b***

This is the standard relation between procedures, which holds if and only if *a* may call *b*.

• ***a* creates *b***

This relation holds if and only if *a* may create *b*. It exists in a language or systems where processes can start other processes (e.g. Ada tasks, Unix processes, PL/I tasks, Simula classes). The same relation also applies to the case where *b* is a data structure in languages where data can be allocated dynamically (e.g. **new** in Pascal).

• ***a* activates *b***

This relation holds in systems supporting coroutines (e.g. Simula) or parallel processes (Ada) if and only if if *a* may re-start a suspended execution of *b*.

• ***a* sends_information_to *b***

This relation holds if and only if *a* may pass information to

b. Let *receives_information_from* be the inverse relation. The following constraint holds for common programming languages:

sends_information_to ∪ *receives_information_from* ⊆ *calls* ∪ *is_called_by*

However, this is not the case in CSP, for example, where information may also be passed through the "rendez-vous" mechanism; thus in these systems:

sends_information_to ∪ *receives_information_from* ⊆ *calls* ∪ *is_called_by* ∪ *activates*

More Constraints

Many features of programming languages may be characterized as properties of the above relations. For example, defining

same_scope = *declared_in* ; *has_declaration*

then in block-structured languages such as Algol 60:

refers_to ⊆ *declared_in* * ∪ *same_scope*

but in Ada:

refers_to ⊆ *declared_in* * ∪ *same_scope* ∪ (*declared_in* ; *refers_to*)

In all common languages, we have

calls ∪ *creates* ∪ *activates* ⊆ *uses* etc.

5.6. Time and system consistency

For the purpose of this study, only one property of the basic type Time matters: the fact that it is totally ordered by a relation which we call *before*. The inverse relation is predictably called *after*.

As mentioned in section 3.2, we define *time_of_last_change* as an attribute rather than a relation. This is merely for convenience; mathematically, an attribute is a (possibly partial) function, thus a special case of a relation anyway. Let *changed_at* be the inverse of *time_of_last_change*.

Part of the problem of configuration management is due to the fact that no element in a system should be younger than any element which depends on it. This is expressed by the following constraint, which we may call the fundamental law of system consistency:

changed_at ; *depends_on* ; *time_of_last_change* ⊆ *after*

where relation *depends_on* is defined as:

depends_on = *contains* ∪ *instances* ∪ *generalizes* ∪ *refers_to* ∪ *needs*

5.7. Relations between program elements

Our last set of relations will contain relations between objects of a program. These relations play an essential role in static program analysis, whether it is for compiler optimization, supercomputer programming[7,23], or program debugging.

- ***a* follows *b***

This relation holds if and only if *a* is a statement whose execution may be immediately followed by that of statement *b*. It describes the flow of control.

- ***a* accesses *b***

This relation holds if and only if *a* is a statement or a program module, *b* is a program object (variable, etc.), and the value of *b* is needed for the execution of *a*. For example, if *a* is an assignment statement, it **accesses** the objects on the right-hand side.

- ***a* modifies *b***

This relation holds if and only if *a* is a statement or a program module, *b* is a program object (variable, etc.), and the value of *b* may be modified during the execution of *a*. For example, if *a* is an assignment statement, it **modifies** the variable on the left-hand side.

- ***a* needs_value_of *b***

This relation holds if and only if *a* and *b* are objects of a program (e.g. variables), and the value of *a* may be modified by a computation which uses the value of *b*.

The following constraint may be called the fundamental law of static analysis:

needs_value_of ⊆ (*modifies* $^{-1}$; (*follows* ; *modifies*) ∩ *accesses*)* ; *modifies* $^{-1}$; *accesses*

To understand this constraint, it may be useful to look at figure 2, where *i* and *j* are statements, and *a, b, c* are program objects, and to note that the solution of

$d = r \cup (t ; d)$

is

$d = r \cup (t ; r) \cup (t ; t ; r) \cup (t ; t ; t ; r) \cup \ldots$

i.e

$d = t^* ; r$

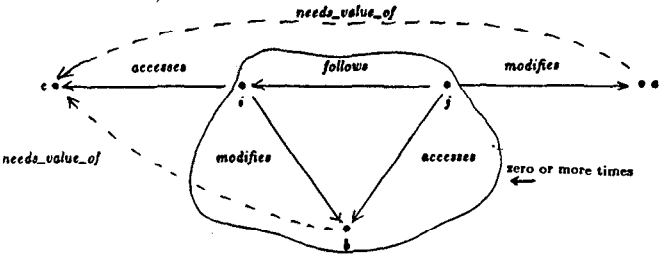

Figure 2: The Static Analysis Constraint

6. STATE OF THE SYSTEM

After the initial specification and design phase, the SKB project currently (June 1985) pursues the following tasks:

- The CRAC calculus has been defined precisely[13] and is being further refined to include diverse kinds of object manipulation and user queries.
- A prototype has been implemented in Prolog[27]; an alternative approach, using the relational data base management system Ingres, is pursued concurrently. An experimental graphical interface (the "designer's sketchpad" mentioned in section 4) is also being implemented.
- The study of useful software relations outlined in section 5 of this paper is being further refined.
- Two unrelated software projects, one at UC Santa Barbara and one in industry, have been the object of an in-depth analysis[19] with two complementary aims: to assess practitioners' needs from their current practices, and to evaluate the CRAC as a modeling tool.
- Finally, efficient multigraph algorithms for the incremental monitoring of constraints have been investigated[13].

Acknowledgments

Several UCSB students contributed useful ideas, notably Jacques Delort, Xavier Glikson and Lucio Mendes. The referee's comments were helpful. I also thank Peter Lohr for several important observations.

References

1. Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, "A Specification Language," in *On the Construction of Programs*, ed. R. McNaughten and R.C. McKeag, Cambridge University Press, 1980.
2. Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading (Massachusets), 1979.
3. Mack W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 60-68, January 1977.
4. Dines Bjørner and Cliff B. Jones, *Formal Specification and Software Development*, Prentice-Hall, Englewood Cliffs (New-Jersey), 1982.
5. Barry W. Boehm, "Software Engineering - As It Is," in *Proceedings of the 4th International Conference on Software Engineering, Munich (Germany)*, pp. 11-21, IEEE, September 1979.
6. Barry W. Boehm, Maria H. Penedo, E. Don Stuckle, Robert D. Williams, and Arthur B. Pyster, "A Software Development Environment for Improving Productivity," *Computer (IEEE)*, vol. 17, no. 6, pp. 30-44, June 1984.
7. Alain Bossavit and Bertrand Meyer, "The Design of Vector Programs," in *Algorithmic Languages*, ed. Jaco de Bakker and R.P. van Vliet, pp. 99-114, North-Holland Publishing Company, Amsterdam (The Netherlands), 1981.
8. G. Bracci, P Padini, and G. Pelagatti, "Binary Logical Associations in Data Modeling," in *Modeling in Data Base Management Systems, IFIP Working Conference on Modeling in DBMS's*, ed. G.M. Nijssen, 1976.
9. Ronald J. Brachman, "What IS-A and isn't: An Analysis of Taxonomic Links in Semantic Networks," *Computer (IEEE)*, vol. 16, no. 10, pp. 67-73, October 1983.
10. John Buxton, *Requirements for Ada Programming Support Environments: Stoneman*, US Department of Defense OSD/R&E, Washington, D.C., February 1980.
11. S. Ceri and Stefano Crespi-Reghizzi, "Relational Data Bases in the Design of Program Construction Systems," *SIGPLAN Notices*, vol. 18, no. 11, pp. 34-44, November 1983.
12. Ian D. Cottam, "The Rigorous Development of a System Version Control Program," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 2, pp. 143-154, March 1984.
13. Jacques Delort, "The Calculus of Relations and Constraints: Definition and Algorithms," Forthcoming Master's Thesis, University of California, Santa Barbara, 1985.
14. Jacky Estublier and Said Ghoul, "Un Système automatique de gestion de gros logiciels, la Base de Programmes Adèle / An Automated Management System for Large Software, the Adele Data Base," *TSI (Technique et Science Informatiques / Technology and Science of Informatics)*, vol. 3, no. 4, pp. 253-260 (French edition), 221-240 (English Edition).
15. Stuart I. Feldman, "Make - A Program for Maintaining Computer Programs," *Software, Practice and Experience*, vol. 9, pp. 255-265, 1979.
16. Lloyd D. Fosdick and Leon J. Osterweil, "Data Flow Analysis in Software Reliability," *Computing Surveys*, vol. 8, no. 3, pp. 305-330, 1976.
17. Michael T. Garrett and James D. Foley, L. P. Deutsch, and B. W. Lampson, "An online editor," *Comm. Assoc. Comp. Mach.*, vol. 10, no. 12, pp. 793-799, 803, December 1967.
18. Susan L. Gerhart, "Application of Axiomatic Methods to a Specification Analyzer," in *Proceedings of the 7th International Conference on Software Engineering*, pp. 441-451, ACM-IEEE Computer Society, Orlando (Florida), March 26-29, 1984.
19. Xavier Glikson, "Analysis and Modeling of Software Configuration Management Practices," Forthcoming Master's Thesis, University of California, Santa Barbara, 1985.
20. Nico Haberman et al., *The Second Compendium of Gandalf Documentation*, Carnegie-Mellon University, Pittsburgh (Pennsylvania), 1982.
21. Cliff B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs (New-Jersey), 1980.
22. Ken Kennedy, *Automatic Translation of Fortran Programs to Vector Form*, Rice University, Department of Mathematical Sciences, October 1980.
23. David J. Kuck, R. H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, *Compiler Transformation of Dependence Graphs*, Conference Record of the Eighth ACM Symposium on Principles of Programming Languages, Williamsburg (Virginia), January 1981.
24. Mark A. Linton, "Implementing Relational Views of Programs," in *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ed. Peter Henderson, pp. 132-140, Pittsburgh, Pennsylvania, April 23-25, 1984. Appears as Software Engineering Notes 9, 3 (May 1984) and SIGPLAN Notices 9, 3 (May 1984).
25. Alan K. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence*, vol. 8, pp. 99-118, 1977.
26. Bruce J. McLennan, "Overview of Relational Programming," *SIGPLAN Notices*, vol. 18, no. 3, pp. 36-44, March 1983.
27. Lucio Dimas dos Santos Mendes, "A Prolog Implementation of the Software Knowledge Base," Forthcoming Master's Thesis, University of California, Santa Barbara, 1985.
28. Bertrand Meyer, "Towards a Relational Theory of Software," Internal Report, University of California, Santa Barbara, July 1984.

29. Bertrand Meyer, "A System Description Method," in *International Workshop on Models and Languages for Software Specification and design*, ed. Robert G. Babb II and Ali Mili, pp. 42-46, Orlando (Fl.), March 1984.

30. David L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 2, pp. 128-138, March 1979.

31. Naphtali Rishe, "A Relational Database Design Methodology Using Binary Conceptual Schemata," Technical Report, Department of Computer Science, University of California, Santa Barbara, January 1985.

32. Mark J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364-369, December 1975.

33. Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, MIT Press, Boston, Massachussets, 1977.

34. Daniel Teichroew and Ernest A. III Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 16-33, January 1977.

35. Kim Walden, "Automatic Generation of Make Dependencies," *Software, Practice and Experience*, vol. 14, no. 6, pp. 575-585, June 1984.

APPENDIX

RELATIONS

Let X and Y be two sets. The set of binary relations (or just relations) between X and Y, denoted $X \longleftrightarrow Y$, is defined as the powerset (set of subsets) of the cartesian product $X \times Y$:

$$X \longleftrightarrow Y = \mathbf{P}\,(X \times Y)$$

In other words, a relation r between X and Y, i.e. an element of $X \longleftrightarrow Y$, is a set of pairs

$$\{ [x_1, y_1], [x_2, y_2], \ldots \}$$

with $x_i \in X$ and $y_i \in Y$ for all i.

As a notational convention, the sets of interest (those between which relations are defined) will have names beginning with upper-case letters, e.g. X, *Specification*, etc. Names of set elements and those of relations will be written in lower-case, e.g. x, r, *part_of*. To express that a certain pair of elements $x \in X$, $y \in Y$ belongs to a relation r, i.e. that

$$[x, y] \in r$$

it is often convenient to use an infix notation, as in

$$x \ \mathbf{uses}\ y$$

(where x and y might be program modules), rather than

$$[x, y] \in \mathit{uses}.$$

We will use the convention that the name of a relation, written in **boldface** as in this example, may be used as an infix operator.

Since any relation in $X \longleftrightarrow Y$ is a subset of $X \times Y$, we can talk about the intersection of two relations, denoted $r \cap s$, and their union, denoted $r \cup s$. We may also express that a relation is included in (is a subset of) another, by writing $r \subseteq s$.

The inverse of relation $r \in X \longleftrightarrow Y$ is the relation r^{-1} in $Y \longleftrightarrow X$ such that

$$y\ r^{-1}\ x \iff x\ r\ y$$

The domain of $r \in X \longleftrightarrow Y$, written *domain (r)*, is the subset of X containing all elements x for which $x\ r\ y$ holds for some $y \in Y$. The range of r, written *range (r)*, is *domain* (r^{-1}).

The composition of two relations $r \in X \longleftrightarrow Y$ and $s \in Y \longleftrightarrow Z$, written $s \bullet r$, is that relation in $X \longleftrightarrow Z$ which holds between elements x and z if and only if

$$x\ r\ y \text{ and } y\ s\ z \text{ for some } y \in Y$$

The order of the arguments to the composition operator is traditional in mathematics and has some justification; to many people, however, it is less confusing to write the relations in the order in which they are "applied"; thus rather than the dot notation we use the semi-colon notation, with $r\ ;\ s$ being defined as $s \bullet r$ (the use of the semi-colon is justified by the close connection which exists between statement sequencing in programs and composition of relations; see[4]).

For any set X, the identity relation on X, denoted *id (X)*, or just *id* when there is no ambiguity, is the "diagonal" relation which holds only between each element and itself. We call *null* the empty relation.

Let $r \in X \longleftrightarrow X$ (source and target set identical). The successive powers of r are defined as follows:

$$r^0 = id$$
$$r^i = r\ ;\ r^{i-1} \qquad (i > 0)$$

A relation $r \in X \longleftrightarrow X$ has a transitive closure, denoted r^+, and a reflexive transitive closure, denoted r^*, defined as follows:

$$r^+ = r \cup r^2 \cup r^3 \cup \ldots$$
$$r^* = id \cup r^+$$

A relation $r \in X \longleftrightarrow X$ is:

- Transitive iff $r^2 \subseteq r$ (or equivalently $r^+ = r$)
- Reflexive iff $id \subseteq r$
- Symmetric iff $r^{-1} = r$
- Antisymmetric iff $r \cap r^{-1} \subseteq id$
- Functional iff $r^{-1}\ ;\ r \subseteq id$ (note that this characterizes partial functions)
- Total iff $id \subseteq r\ ;\ r^{-1}$

A (partial) order is a transitive, antisymmetric and reflexive relation. Such an order relation is total if and only if $r \cup r^{-1} = X \times X$.

A dag (directed acyclic graph) is a relation r such that r^* is a partial order. A dag is rooted if and only if, for any $y \in X$, the set of $x \in X$ such that $x\ \mathbf{r}^*\ y$ is finite; a root is then an element of X - *domain* (r^{-1}). It is easily shown that in a rooted dag, for any $y \in X$, there is at least one root x such that $x\ \mathbf{r}^*\ y$.

A forest is a rooted dag r such that r^{-1} is functional (note that r represents the relation between parent and child). It is easily shown that a non-empty forest has at least one root. A tree is a forest with at most one root.