THE ANALYSIS OF MODULAR STRUCTURES WITH RESPECT TO THEIR INTERCONNECT TOPOLOGY

PIETER H. HARTEL and FRANK A. VAN HARMELEN

Vakgroep Informatica, Universiteit van Amsterdam, Amsterdam (The Netherlands)

(Received October 25, 1983)

Summary

A comparison of hardware and software interfaces shows some differences between them; these differences may explain why it is harder to develop modular software than modular hardware.

A mathematical model of a software interface interconnection topology is developed to provide a better understanding of the mechanisms involved in modularization.

1. Introduction

One of the most difficult problems in software engineering is how to partition a complex task into a manageable number of subtasks such that the job may be done by a number of people without sacrificing conceptual integrity [1]. Once a description has been given of what the subtask consists of (*i.e.* the interface of the module to the rest of the world has been *completely* specified), the implementation and testing can be done autonomously.

Both for hardware and for software development, techniques are available to specify interfaces. For example, in hardware development, formal specification methods (*e.g.* ISPS) are used to aid in the simulation of the circuitry being designed. However, in general the development methods available are informal and incomplete. The degree to which a design is truly modular and hierarchical depends very much on the talent and experience of the architect and the tools available. As long as the designer is present during the lifetime of his product, the situation will be satisfactory because the architect can always be asked. However, a more common situation is where a product arrives which does not quite provide the required services [2] or, even worse, which does not function at all because it was intended for a different hardware and/or software environment. For the programmer given the task of sorting out the problem, any information is welcome, and the availability of the original design documents will be especially appreciated.

In this paper a method will be developed to extract some of this information automatically from a program text. This is done by representing the modular structure of a program in a graph which is suited to automatic manipulation.

2. Hardware interfaces

In hardware technology, interfaces may be described at various levels. A typical chip has a number of pins to which voltages may be applied. As a result, currents may be drawn from other pins to drive connected circuitry. All voltages and currents are within certain ranges (or the circuit will blow up) and have precise timing constraints. A data sheet either implicitly or explicitly (*e.g.* for TTL circuitry) describes both the incoming and the outgoing signals completely, albeit informally. Out of these components, larger-scale modules which are typically on printed circuit boards are built. The description of a module at the signal level is necessarily very similar to that of its components. A module as used in computing equipment usually interfaces to a common bus which transports signals between modules. With each bus a protocol is defined to guarantee an orderly use of the transport capacity. The description of an interface of a module involves the specification of signal levels, timing constraints, bus protocols and a functional description of which pin of the interface serves what purpose. Again both incoming and outgoing signals are described. These properties of hardware modules make it possible to build systems out of modules acquired from different sources; from an economic point of view this is very attractive.

2.1. Interconnection topology

Hardware interfaces mostly deal with simple topological structures. Many modules interface to the same bus. Some modules merely serve to connect busses together. A typical interconnection topology would be as in Fig. 1. The busses are drawn as horizontal lines, whereas the modules

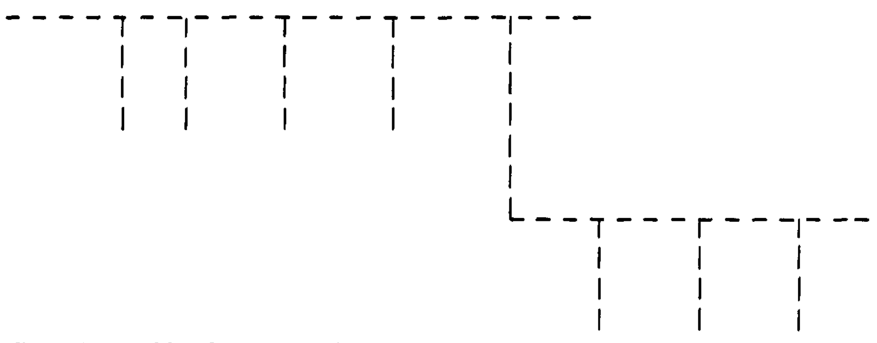

Fig. 1. Typical hardware module interconnection topology.

connected to the busses are drawn as vertical lines. The simplicity of the interconnection topology facilitates modular hardware design and maintenance. Designing the interfaces themselves remains a difficult problem.

Part of the success of Digital Equipment Corporation's PDP-11 computer family is due to its modular design around a flexible asynchronous bus. Functionally equivalent (*e.g.* memory modules) but electrically very different modules may be mixed freely and used simultaneously.

3. Software interfaces

In programming, modularization through separate compilation is possible since the advent of FORTRAN in the late 1950s. Only more recently has the meaning of separate compilation changed to separate-but-not-independent compilation [3]. The design of many modern programming languages (*e.g.* ADA, Modula-2) reflects this change. These together with the notions of strong typing and abstract data types are the most powerful tools in current programming methodology, but are they really powerful?

The *user* of a module, be it a programmer or another module, should not need to know the implementation details. These are therefore best (physically) separated from the interface specification. Unfortunately, practical techniques for the semantic specification of interfaces for most non-trivial programming projects are currently non-existent. Consequently specifications are purely syntactical and the semantics are described (if at all) in an informal way (*e.g.* packages in ADA, or modules in Modula-2). It is neither practical nor possible for the user of a module to study its implementation. Until full semantics are included in interface specifications, other mechanisms are required to maximize modular independence.

3.1. Interconnection topology

Software interfaces are part of complex structures. Such interfaces are designed on an *ad hoc* basis and are frequently more geared to making the task of the implementor easy than to making the module simple to use. A program can be thought of as being made up of a number of modules arranged in a certain hierarchy. The "main loop" of a program would be at the top level of the hierarchy, whereas the machine-dependent modules would be at the lowest level.

The interconnection topology of a program can be represented as a graph (Fig. 2)*. The vertices represent separate modules whereas the edges

*This configuration corresponds to a stand-alone real-time program which ran on a microcomputer. The microcomputer was connected to a number of terminals on one side and to a larger number of host computer ports on the other. The main purpose of the system was to allow for multiple interactive sessions to be in progress concurrently from a single terminal.

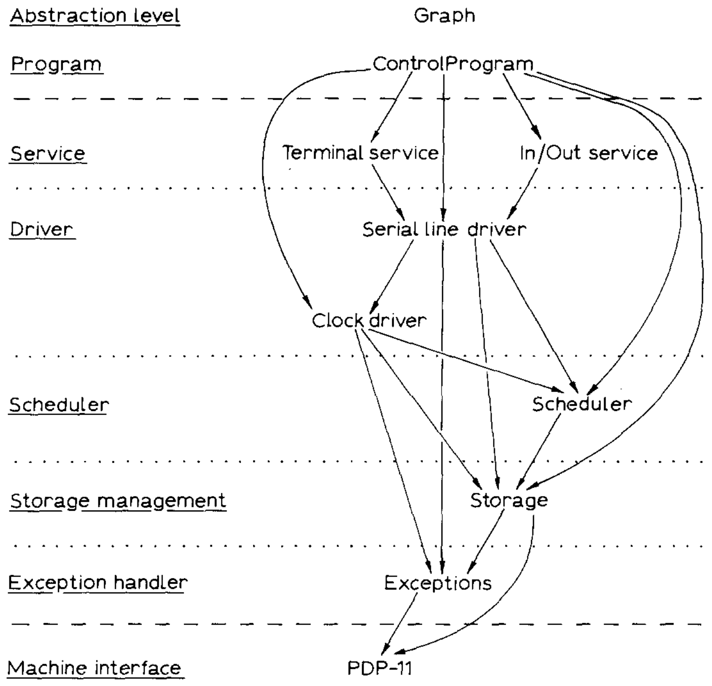

Fig. 2. Typical software module interconnection topology.

represent their interconnections. If a module *a* references a module *b*, the direction of the corresponding edge in the graph will be from *a* to *b*. Strictly speaking only three levels of abstraction can be distinguished: the top level with the control program, the bottom level with the PDP-11 module and the intermediate level with all other modules. This is indicated by the broken lines. Functionally, seven levels of abstraction can be distinguished as indicated in the column on the left of Fig. 2.

An intriguing question is why a design is not strictly hierarchical. An equivalent question would be why services are provided at a lower level of abstraction and used directly at much higher levels rather than being "passed on" through intermediate levels*.

*If efficiency is of limited concern, there may be no penalty for introducing intermediate levels, but in particular in real-time applications this is not always true. In an early version of this example program, interrupts from serial lines were intercepted by the scheduler and dispatched to the appropriate driver. This introduced just enough overhead to cause the program to lose interrupts at busy times. After the interrupt interception facility (provided by the language–machine interface at the very lowest level of abstraction) was moved to the driver level, no interrupts were lost any more.

In order to find an answer to such questions, a more strict formalism is needed to describe the graph, and a representation is needed that is suited to automatic manipulation.

4. The model

Let V be the set of modules of a software system, and $n = |V|$ the number of modules. Let us define a relation $[R]$ on $V*V$ by

$$x[R]y \Longleftrightarrow \text{module } x \text{ makes a direct reference to module } y \qquad (1)$$

for all x, y in E. $[R]$ can be represented by its corresponding graph $G = (V, E)$. V is the set of vertices of G and E the set of edges as defined by

$$(x, y) \text{ in } V \Longleftrightarrow x[R]y \qquad (2)$$

for all x, y in V. The graph illustrated in Fig. 2 corresponds exactly to these definitions.

In the following the graph G is assumed to be connected. This is not a limitation since an unconnected graph would represent several separate software packages, and each package could be treated separately.

Various interesting properties of the modular structure represented by G can be derived. For this purpose, A is defined as the incidence matrix of G. A is a square matrix of order n. Each row and each column are associated with a module, one row and one column per module. The elements of A are defined by

$$\begin{array}{ll} \mathrm{A}[i, j] = 1 & \text{if there is an edge from vertex } i \text{ to vertex } j \\ & \text{(module } i \text{ references module } j) \qquad (3) \\ \mathrm{A}[i, j] = 0 & \text{otherwise} \end{array}$$

A can be interpreted as follows. Each 1 in row x of A indicates a reference from module x to another module, and each 1 in column y indicates a reference from another module to y (Fig. 3).

It is important to know whether the graph G contains cycles or not. For this purpose let us define A* as the transitive closure of A. Then

$$\begin{array}{ll} \mathrm{A}^*[i, j] = 1 & \text{if there is a path from vertex } i \text{ to vertex } j \\ & \text{(module } i \text{ directly or indirectly references module } j) \qquad (4) \\ \mathrm{A}^*[i, j] = 0 & \text{otherwise} \end{array}$$

An example of A* is given in Fig. 4. A* contains information concerning possible cycles in G:

$$G \text{ is cycle free} \Longleftrightarrow \text{for all } i \text{ in } 1, \ldots, n\text{:} \quad \mathrm{A}[i, i] = 0 \qquad (5)$$

(all diagonal elements of A* are zero, *i.e.* there is no path from any module to itself).

	Scheduler	Clock driver	Exceptions	In/Out service	PDP-11	Control program	Serial line driver	Storage	Terminal service
Scheduler	0	0	0	0	0	0	0	1	0
Clock driver	1	0	1	0	0	0	0	1	0
Exceptions	0	0	0	0	1	0	0	0	0
In/Out service	0	0	0	0	0	0	1	0	0
PDP-11	0	0	0	0	0	0	0	0	0
Control program	1	1	0	1	0	0	1	1	1
Serial line driver	1	1	1	0	0	0	0	1	0
Storage	0	0	1	0	1	0	0	0	0
Terminal service	0	0	0	0	0	0	1	0	0

Fig. 3. Incidence matrix corresponding to Fig. 2.

	Scheduler	Clock driver	Exceptions	In/Out service	PDP-11	Control program	Serial line driver	Storage	Terminal service
Scheduler	0	0	1	0	1	0	0	1	0
Clock driver	1	0	1	0	1	0	0	1	0
Exceptions	0	0	0	0	1	0	0	0	0
In/Out service	1	1	1	0	1	0	1	1	0
PDP-11	0	0	0	0	0	0	0	0	0
Control program	1	1	1	1	1	0	1	1	1
Serial line driver	1	1	1	0	1	0	0	1	0
Storage	0	0	1	0	1	0	0	0	0
Terminal service	1	1	1	0	1	0	1	1	0

Fig. 4. Transitive closure of the incidence matrix.

The relation $[R]$ in eqn. (1) implicitly defines a partial ordering on V. A topological sort can now be applied to G in order to put the modules in a sorted list. The sorting algorithm is as follows [4].

(I) Create an empty list.

(II) Look for all vertices that have no outgoing edges.

(III) Add these vertices (modules) to the sorted list.

(IV) Remove these vertices and their incoming edges from G, giving G'.

(V) Apply steps (II) - (V) from the algorithm to G'.

This algorithm defines a total ordering on the modules. A can now be rewritten such that the rows and columns will appear according to this ordering, giving the matrix A_s. A_s is an upper triangular matrix, with all the diagonal elements zero. Figure 5 is an example of such a sorted matrix.

	Control program	Terminal service	In/Out service	Serial line driver	Clock driver	Scheduler	Storage	Exceptions	PDP-11
Control program	0	1	1	1	1	1	1	0	0
Terminal service	0	0	0	1	0	0	0	0	0
In/Out service	0	0	0	1	0	0	0	0	0
Serial line driver	0	0	0	0	1	1	1	1	0
Clock driver	0	0	0	0	0	1	1	1	0
Scheduler	0	0	0	0	0	0	1	0	0
Storage	0	0	0	0	0	0	0	1	1
Exceptions	0	0	0	0	0	0	0	0	1
PDP-11	0	0	0	0	0	0	0	0	0

Fig. 5. Sorted incidence matrix corresponding to Fig. 2.

It is possible that certain iterations of the algorithm yield more than one candidate for the list. In this situation the sorting algorithm does not uniquely define the total ordering of the modules; all the modules that are found in step (II) of a particular iteration form one group and can be placed in an arbitrary order. However, this does not affect the upper triangular form of A_s. All modules that belong to one group can be thought of as belonging to the same level in the hierarchy defined by G. For example in Fig. 2 the terminal service and in/out service modules belong to the same group. To each module x in a group we assign a number $\|x\|$, its abstraction level. Groups that are higher in the hierarchy (groups that are found in a later iteration of the algorithm) have higher abstraction levels.

A hierarchy is strict if, for all x, y in V

$$(x, y) \text{ is in } E \Rightarrow \|x\| - \|y\| = 1 \tag{6}$$

(if one module references another, its abstraction level must be one level higher). It is possible to verify whether G is strict or not. Let us define the corresponding matrix A_l. For each abstraction level, A_l has one row and one column, whereas

$$\begin{aligned} A_l[i, j] &= 1 \quad \text{if a module of abstraction level } i \text{ references a module of} \\ &\qquad \text{abstraction level } j \\ A_l[i, j] &= 0 \quad \text{otherwise} \end{aligned} \tag{7}$$

Figure 6 gives an example of this.

G is strict if A_l only has elements 1 on its upper secondary diagonal and nowhere else. With A_l it is possible to measure the degree to which G is strict. The number of elements 1 not on the upper secondary diagonal represents the amount of non-strict references in G, and the distance of a 1 to the diagonal is the number of levels that such a reference crosses. For

$$\begin{array}{l|cccccccc}
 & \text{Control program} & \text{In/Out \& Terminal} & \text{Serial line driver} & \text{Clock driver} & \text{Scheduler} & \text{Storage} & \text{Exceptions} & \text{PDP-11} \\
\text{Control program} & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\text{In/Out \& Terminal} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\text{Serial line driver} & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
\text{Clock driver} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\text{Scheduler} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\text{Storage} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{Exceptions} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\text{PDP-11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Fig. 6. Abstraction level matrix.

example the reference in Fig. 2 from the control program to the scheduler corresponds to element [1, 5] in Fig. 6.

Another interesting property of a graph G is whether it is a tree or not. G is a tree if each vertex except one (the root) has exactly one connecting edge. This means that each module is only referenced by one other module. As a consequence each row of the corresponding matrix A_S, except the first row, contains exactly one 1.

There is a problem if G contains cycles. In such a situation the abstraction levels of modules in a cycle are not well defined. Let us consider the example in Fig. 7. This example implies the impossible case where $\|x\| > \|y\| > \|z\| > \|x\|$. One solution is to assign the same abstraction level to all modules in a cycle, although this sometimes leads to undesired conclusions about the strictness of a hierarchy. This is illustrated in Fig. 8. In

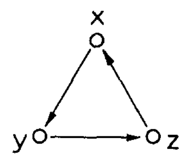

Fig. 7. Abstraction levels in a cycle.

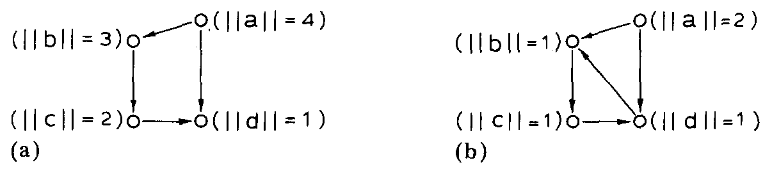

Fig. 8. Two configurations of abstraction levels.

Fig. 8, neither configuration is strict. However, if in Fig. 8(b) the modules *b*, *c* and *d* were considered as one, then the resulting hierarchy would be strict.

5. Implementation

The model developed in Section 4 has been applied to a number of "real world" software systems. It has proved to provide interesting and useful information.

The type of information to be processed may be extracted automatically from program modules written in languages such as ADA and Modula-2. It is sufficient to parse the source text of the modules for reference information (*i.e.* USE and IMPORT clauses). Three Modula-2 systems were readily available in source form to be processed according to the model.

The actual implementation of the algorithms described in Section 4 consists of a program which scans source modules to discover which modules are referenced directly. This information is fed to a topological sort program and converted into the desired matrix representation, using Unix [5] utilities (sed, awk, tsort and sort) (Unix is a trademark of Bell Laboratories).

5.1. Constructing the incidence matrix

During the construction phase of the incidence matrix, modules may be discovered to which references are found but which themselves are not found. This should be interpreted as an early warning to the programmer that at some stage either the missing modules will have to be coded or the originator of the software system will have to be asked for the missing items. Without this warning, most of the programming and adaptation effort would have been completed before it was discovered that one or more modules were missing (*e.g.* during system generation or linkage). Such omissions manifest themselves as missing rows in the incidence matrix. The three software systems processed had one missing module out of a total of 300.

The matrix may or may not be square, because there may be columns missing as well; this is what would happen if a module makes references to other modules but is not referenced by any module itself. Let us consider for example the module containing the main loop of a program. It will not be referenced by other modules (at least not explicitly). The case where more than one module is not referenced is more intricate. This might indicate the presence of obsolete modules, modules included for test purposes or modules that are present in different versions of the system (for instance using conditional compilation).

In order to make the matrix square, either an empty column will have to be introduced in the matrix or the row must be deleted.

5.2. Interpreting the results

Cycles in the graph of a software system cause problems. All modules in a cycle have to be treated simultaneously during design, implementation, testing and maintenance. Therefore their presence spoils the modular structure of a software system. The computation of the A* matrices corresponding to the three software packages unveiled the presence of some cycles, but these were all too small (at most three modules) to disturb the hierarchy in a significant way.

The A* matrix shows which modules are directly or indirectly dependent on other modules. This information can be used to select a testing strategy. All modules that reference a modified module will have to be retested. The order in which testing takes place is determined by the abstraction level of the modules. The example in Fig. 4 shows the unfortunate case where any modification would have to be followed by a test of almost all modules at higher levels of abstraction.

In practice, most software packages are not strictly hierarchical. This impairs the portability and maintainability of software packages.

6. Conclusions

The aspects of the differences between hardware and software design that concern the interconnect topology of a system of modules were discussed. A model was developed to provide a basis for the evaluation of modular software. The viability of the model was tested by applying it to a small set of software systems. Although many useful facts about these systems were derived, some questions remain.

The usefulness of the model could be improved if more information about the actual interfaces could be included in the model. Connections between modules could be weighted, depending on the degree to which one module relies on the other [6].

In a large set of modules, subsets may exist which together provide a service to the remaining modules. A mechanism to discover such subsets would be useful for the evaluation and maintenance of software systems.

Acknowledgments

We would like to thank L. O. Hertzberger and P. Klint for their many helpful comments.

References

1 F. P. Brooks, Jr., *The Mythical Man-month, Essays on Software Engineering,* Addison-Wesley, Reading, MA, 1975.

2 P. H. Hartel, Comparing Pascal and Modula-2 as systems programming languages. In J. Bormann (ed.), *Proc. International Federation for Information Processing TC2 Working Conf. on Programming Languages and System Design,* North-Holland, Amsterdam, 1983.

3 N. Wirth, *Programming in Modula-2*, Springer, Berlin, 1982.
4 D. E. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, MA, 1967.
5 D. M. Ritchie, The UNIX time-sharing system, *Commun. ACM, 17* (7) (1978) 365 - 375.
6 F. M. Haney, Module connection analysis — a tool for scheduling software debugging activities, *Proc. 1972 Fall Joint Computer Conf.*, Vol. 41, Part 1, Afips Press, Montvale, NJ, 1972, pp. 173 - 179.