More on simulation languages and design methodology for computer systems*

by DAVID L. PARNAS

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

In an earlier paper¹ we attempted to set forth (1) a design methodology for computer systems which made heavy use of simulation and (2) a simulation language intended to facilitate the use of the design methodology presented. The basic justification for the design methodology presented was an old precept from engineering design: a problem must be defined before it is solved. The result was a methodology which laid great stress on specifying the behavior of a system or a component in a system before producing the design. The simulation language, SODAS, was designed to allow a design to proceed in a hierarchical way, treating any system as a set of components, specifying the behavior of those components, then treating the components themselves as systems. By means of the SODAS language it was to be possible to evaluate the design at any stage in its development without excess effort.

One of the most fruitful results of the publishing of "SODAS . . ."¹ has been a number of useful discussions on design methodology and simulation with other workers. Probably the most fruitful of these discussions has been with Brian Randell, who had been interested in similar problems, but with a somewhat different emphasis and result.³ A major result of these discussions has been the realization that while the basic design methodology described in "SODAS . . ."¹ is most general and should apply to all sorts of computer system design problems, the specific characterization of it given in "SODAS . . ."¹ itself fails to meet such application requirements.

An examination of the work leading to and motivating SODAS shows that all the motivating examples fall into a rather restricted class of hardware modules. That class can be roughly characterized as single level*, involving no interpretation of programs or even microprograms. It is not surprising then that we find that the particular description of the design methodology given in "SODAS . . ."¹ and the structure of SODAS itself are fairly closely restricted to that class of systems.

It is our purpose in this paper to explore a somewhat less restricted expression of our basic design methodology in an attempt to extend it beyond the limited confines covered by "SODAS . . ."¹. In particular, we expect to explore the design of computer systems consisting of at least two (and often more) levels of hardware and software. We shall refer to our system class as the design of Operating Computer Systems (OCS), since it usually includes both hardware and the software known as the operating system for the hardware. It is also our hope to take into account the concepts about multilevel simulation given in Zurcher and Randell,² working them into the design of a sim-

* It is necessary to note here that we are deviating from the way that level is used in "SODAS . . ."¹ to a usage that is more consistent with Zurcher and Randell.² In "SODAS . . .",³ level referred to level of detail, i.e., the number of levels of definition design completed in a state of partial design. The use of level here is somewhat more complex and will be defined more precisely in the text.
ulation language which is more broadly applicable than SODAS.**

The principal difference between the class of systems discussed in "SODAS . . . ."1 and the class that we are now interested in arises from the existence of a program known as the operating system, which should be considered an integral part of the design problem. Early computers were designed with no thought of an operating system, simply because they were to be run without an operating system. As the need for an operating system became apparent, these were designed separately from the fixed piece of hardware and had as their aim efficient and convenient use of that piece of hardware.

The premise that we should proceed by specifying the behavior of a system before designing its components implies that we can no longer look at an operating system as an item to be placed on a previously designed piece of hardware. The actual design should begin with a specification of the overall behavior of the hardware/software combination. It continues by dividing the system into components and they, in turn, are designed with little or no attention to the question of what will be hardware and what will be software until very late in the design.

In the type of system that SODAS deals with, the nature of the system building block was clear; each hardware module is composed of smaller or lower level hardware modules until one gets to the realm of logic elements. For systems of hardware and software (OCS) we must use as a component or building block a unit which will allow us to postpone the decisions about the hardware/software tradeoff. The unit we have selected is the "sequential process", a rather fuzzy concept which has been discussed elsewhere.3, 4, 5 For our present purposes we note that a sequential process is a fully ordered set of events in an operating computing system and may be performed either by hardware or software. By building a design as a set of such processes we can postpone the hardware/software decision as long as desired.

** We have chosen in this paper to talk not about languages, but about the characteristics or features of languages. We will not at any stage in this paper give a syntax of the language, or even a part of that syntax. Rather, we shall talk exclusively about properties that a language must have for the purposes outlined here. In part this is due to a feeling that the syntax of the language is an irrelevant detail as well as the fact that the syntax of the language is far from frozen at this stage.

Difficulties with SODAS

In the following section we shall attempt to discuss some of the difficulties that the existence of hardware and software as part of a system design introduces into any attempt to use SODAS for such designs.

1. The existence of interconnectors at a lower level (further progress in the design) that did not exist at higher levels (earlier in the design).

   In the hardware modules any wires that connect parts of one subsystem with parts of another subsystem must explicitly connect those two subsystems. As a rule (though not invariably) all significant interconnectors between two components are specified at the time that the functions of those components are specified. No new interconnections show up as the design progresses. In operating systems this is not the case. It is reasonable and desirable that at certain stages in a design the several sequential processes will be described as entirely independent of each other except for certain explicit attempts at communication. At a later stage in the design we will note that these processes have an implicit intercommunication because of resource sharing. In SODAS with its requirement of explicit interconnectors, this would require that the descriptions at the upper level be rewritten. This violates a design criterion for SODAS—the use of SODAS should require no extra effort.

2. The problem of hidden resources.

   At an early stage in the design it will be reasonable to assume that all of a certain class of resource that will ever be required by a process will always be available. At a later stage we decide that for cost reasons it will not. The process will then go to use that resource, find that it is not there and wait until it is available. SODAS will not permit us to add this level of detail without rewriting the description we produced at the more abstract level.

3. Communication through global variables.

   The pretense that all communication is through an explicit set of wires or variables becomes unreasonably restrictive when talking about software operating systems. The simulation language and design methodology must recognize the communication of processes through global variables and files as well as more subtle means of communication (e.g., changing the instruction counter of a process or changing its code).

4. Expansion of time points to time intervals.

   At early days in the design of an OCS certain sequences of computations can be considered to be single events occupying a "point" in time.
and separated by a specified period during which nothing happens. It is assumed that the event cannot be interrupted and that no time passes except between events. At later stages in the design it often becomes necessary to recognize that time passes during an event since conditions not considered at earlier stages of the design may cause the process to be interrupted during an event. This in itself could conceivably be handled within SODAS, or other simulator, but there is no facility that would allow the simulation system to determine the simulated time of this interruption of the event unless we force the programmer to insert timing throughout. The latter requires either recognition of the problems of a later stage of design at too early a stage or substantial modifications to a description which is actually perfectly valid for the level at which it is written.

**Multi-level modeling—two views**

Randell and Zurecher\(^1\) have presented a view of system design and simulation which, on the surface, is quite similar to that presented in "SODAS, etc." Both papers discuss the value of being able to have a simulation model which simultaneously includes descriptions at many levels of detail which are interacting. Close inspection, however, of the two papers indicates that the word 'level' is being used in two different senses. The design process discussed in "SODAS, etc." can be viewed as repeated functional decomposition. Each unit is decomposed into a sub-unit, each with a distinct function. When "SODAS, etc." speaks of 'levels of detail' it is referring to the number of times this decomposition has been carried out in a given unit. Randell\(^2\) sees a different direction of progress for a design, a given functional unit will initially be described as an abstraction from reality. Certain facts about its implementation are initially ignored, while certain design features are being explored. As one recognizes and deals with these facts of life which were initially ignored, one proceeds from a high level of abstraction to a lower level of abstraction. Often in doing a functional decomposition one is simply introducing functional units which are needed to proceed from an abstract description of a component to one which enables the component to exist in a restricted real environment. In proceeding to a lower level of abstraction one is often simply specifying the nature of certain components of the given component. Often, however, the two notions of level are quite different. In functional decomposition the functional units specified are always sub-units of the unit we are decomposing. That unit is always viewed as self contained; in contrast, it is common in going to a lower level of abstraction to recognize sub-units which must be viewed as shared by other components. Thus we are not simply decomposing a given component into its sub-components. Often, too, proceeding to a lower level of abstraction does not involve any functional decomposition at all; it may merely involve taking into account certain external influences on a given component. On the other hand, functional decomposition does not always involve proceeding to a lower level of abstraction. To avoid confusion between these two concepts, we shall hereafter use a “level” to refer to levels of abstraction. “Extent of functional decomposition” will be referred to explicitly, where necessary, by that rather unwieldy phrase.

Both extending functional decomposition and lowering the level of abstraction are consistent with the basic precept of our design methodology; both involve proceeding from a specification of desired behavior to a means of achieving it.

**The nature of distinct levels in OCS**

To date we have left “level” as a fuzzily defined concept. It is necessary now that we determine the nature of levels of abstraction within a complete system, i.e., that we answer the question: Under what circumstances must we consider two actions in a computer system to occur at different levels—or—how do we justify the statement that two subroutines within the system deal with it at a certain level?

A fairly easy answer is that there is no concrete realization of the term level within the system. That the division between levels of abstraction is entirely in the mind of the analyst or designer, who determines what is in the top level by what he chooses to consider first. This rather easy answer, however, ignores a regularity that can be observed if one studies a number of examples of systems so divided into levels.

It appears to be the case that if we have described a set of processes at some level when we proceed to the lower level, we are either specifying or modifying the specification of the interpreters of those processes previously described. In other words, a top level description of a system is a description of a set of processes with an assumed interpreter, that interpreter being essentially passive and accomplishing the actions specified by the process description as directly as possible. When we move down to a lower level, we begin to assign more complex functions to the interpreter, e.g., managing the resources being used by the process and suspending action on the process should insufficient resources be available.

From the collection of the Computer History Museum (www.computerhistory.org)
What SODAS needs

SODAS was designed under the assumption that all the component descriptions and/or process descriptions could be run using the same fixed interpreter. There was no provision for making any substantial alterations to the behavior of the interpreter or for including new ones as part of the design process. Thus SODAS HAS AN INHERENT LIMITATION TO SINGLE LEVEL SYSTEMS.

Each of the difficulties pointed out earlier can be neatly avoided by providing the ability to (a) make certain changes in the operation of an interpreter supplied as part of the system and (b) to provide for the possibility of some simulated processes being the interpreters for others.

SOCS

Having explored a little further the nature of the design process for an OCS, we shall now look at the design of a simulator for use in designing an OCS (SOCS). We can first review the features that we found missing from SODAS.

1. Communication through global variables rather than interconnectors.
2. A flexible interpreter.
3. Ability to simulate a process which is being interpreted by other processes being simulated.
4. Integration of the process describing languages with the network description language.*

We may also list the features that we found in SODAS that were missing in such process oriented languages as SIMULA.**

1. Ability to have a process that consists of a set of processes, e.g., recursive structure.
2. Ability to handle difficult cases of simultaneous events.
3. Ability to handle structural descriptions of hardware.
4. The “wait until” or monitoring feature found in SOL, proposed for SODAS, but missing in SIMULA.

It is not possible at this point to give a detailed description of the SOCS language. It is clear that it will be somewhat reminiscent of SIMULA with extra constructs to provide the features that we have discussed. It is desirable that the language be such that a SIMULA program could be run without substantial changes though we would not object to extensive surface changes. Although not essential, it will probably prove convenient to provide constructions which allow processes to be dealt with as SOL facilities or stores. The sub-languages and connection concept will be carried over from SODAS, though with a somewhat subdued role as it will be possible but not necessary to leave the SOCS language to describe a process (SODAS allowed no primitive process to be described in SODAS itself). At present the problem of simultaneous events looks somewhat less central than it did earlier and will be provided as an optional, run time resolver rather than the compile time solution planned for SODAS.\(^5\)

On following a design philosophy

It is fairly easy to talk about designing from specifications to plans, outside in, top down, or from process to processor. The way that a design should progress is, at least at a certain level of generality, agreed upon. It is quite a different matter to actually carry out a design in that way when dealing with an OCS.

There are two serious difficulties that are encountered should one try to follow this philosophy. The first of these is an inability to really get started, to begin to write down concrete information about the design. One finds oneself in the position of trying to write a program for a machine of unknown characteristics. There is no starting point, no structure to build upon. We are used to designing processes or programs by starting with a very specific tool and looking for a way to take this rather limited tool and get around the limitations. Without the limited tool, with the limitations that we have become accustomed to suddenly gone, we feel lost; the space of possible first steps has become so large that we cannot make the decision.

This difficulty can be somewhat alleviated by providing a higher level language in which to write. We again have restrictions; the syntax and semantics of that language, and the space is again reasonably small. (The space, however, is nowhere near as small as that provided by a machine language and there are very good machine language programmers who still cannot readily get started when faced with a higher level language to write in. This is not a widespread phenomena, however.)

A difficulty which is expected by many people with whom I have discussed the philosophy is the problem of duplicate work. There have been many system designs in which simulation played a part, but it was clearly separate from design work. There was a separate group doing the simulation and it required either extra money or a slower effort by the design group. With talk of really extensive reliance on simulation, people with such experience see a project in which

\(^*\) This refers to the fact that SODAS was a language for connecting components described in other languages rather than a language system.
everything is done twice—once for the simulator and once for the actual system. While some might argue that even such double effort would be worth it if a well designed system were to result, I wish to argue that such double effort can and should be avoided. The code written for the simulator must eventually become part of the running system. If we write a description for a process in some higher level language, such as SOCS, and later specify and design an interpreter for that process, the higher level language will have to be translated into a pseudo-code that is the language of the simulated interpreter. The translator, if the translation is done by a program, and the translated code should both eventually become a part of the system. If, for example, the interpreter is designed directly as hardware, the code is now machine code; if the interpreter becomes a program on some other hardware, it will continue to operate on the same code. The effort involved in producing it for the simulation is not wasted; the product becomes a part of the system that is eventually produced.

Another important aspect of avoiding duplicate work is that it not be necessary to rewrite a description correct at the level of detail at which it was written, because we are progressing further in the design' (We will never avoid rewriting descriptions when we make design changes.) This can be avoided to a very large extent by our ability to modify the behavior of processes, without altering their description, by altering their interpretation or interpreter. The original description thus remains both for purposes of explanation and as a specification of what is expected.

The second real difficulty is encountered at the end of the design process, and has been reported to me by Brian Randell. Randell had been following the design process outlined using a facility for multi-level modeling implemented using FORTRAN running under OS/360. When he reached the point that he felt that the design was complete, he found that it could not become the system, since it still contained many points of dependency on FORTRAN, OS, and the /360 itself. A design done in the way we are discussing cannot be considered complete until the only processes being interpreted by the system interpreter are those simulating the hardware. All other processes must be interpreted by the simulated hardware or by processes being interpreted by the simulated hardware processes, etc. A system like Randell’s which was not specifically designed to allow one process to interpret another, cannot allow the design to proceed to that stage.

Future work

The alert reader will note that while we have been talking about design we have not been designing. It is our intention to design a version of SOCS which can be quickly implemented. We intend to go through a complete design of a system of interest in itself, keeping a protocol of the way that the design progresses. In this way we hope to determine just how accurate our picture of the design process is and further, in what ways SOCS will have to be altered in order to keep from interfering with or distorting the design process. We feel that after using the rudimentary system in a real design, we will have a much better picture of the final version of SOCS.

ACKNOWLEDGMENTS

I wish to acknowledge most gratefully the many hours of discussion that I have had with Brian Randell; each of those hours has improved my understanding of the problems dealt with in this paper. I am also grateful to C. Eastman and A. Perlis, who were kind enough to read early drafts of this paper and to provide many helpful comments.

REFERENCES

1 D L PARNAS J A DARRINGER
SODAS and a methodology for system design
Proc F J C C 1967
2 F W ZURCHER B RANDELL
Multi-level modeling—A methodology for computer system design
Proc International Federation of Information Processing Societies 1968
3 D L PARNAS
Sequential process: A fuzzy concept
Unpublished memo available on request from the author
4 O J DAHL K NYGAARD
SIMULA—An ALGOL based simulation language
C A C M 670-678 September 1966
5 E W DIJKSTRA
Cooperating sequential processes
Technological University, Eindhoven, Netherlands
6 D L PARNAS
Sequential equivalents of parallel processes
Carnegie Institute of Technology Pittsburgh Pa
February 1967