A Systems Design Course Emphasizing Interfaces

Staunstrup, Jørgen

Published in:
Proceedings of MSE'97

Link to article, DOI:
10.1109/MSE.1997.612537

Publication date:
1997

Document Version
Publisher's PDF, also known as Version of record

A Systems Design Course Emphasizing Interfaces

Jørgen Staunstrup
Department of Information Technology, Building 344,
Technical University of Denmark, DK–2800 Lyngby, Denmark*

Abstract

This paper describes an experimental course where students develop a (small) system focusing on the interfaces between different components of the system. The components are developed independently of each other using Web based documentation and focusing on techniques for modeling and analysis of interfaces. These techniques are supported by prototype tools.

1 Introduction

The potentials of the Internet/Intranet as a means for sharing and updating common information has a significant potential for improving communication in large (systems) design projects. This paper describes an experimental systems design course where students develop a (small) system using Web based documentation supported by prototype tools focusing on modeling and checking the interfaces between the different components of a systems design.

Using Web based documentation in an engineering/design environment allows participants to share and update common information such as specifications, design details, and status information. To fully utilize Web based information in a systems design environment it is important to avoid confusion and misinterpretation because of ambiguities. In person to person communication some imprecision in the shared documentation can be tolerated and compensated by direct personal interaction. This possibility is reduced if one relies on Web based (or other written) documentation where lack of precision is not compensated by direct human interaction.

2 Interface design

A common source of errors and delays in design and development projects is misunderstandings caused by inconsistent views of common interfaces. However, insisting that all components have exactly the same view would be too restrictive. It is important to allow them to have different views as long as these are not in conflict. To illustrate this, consider a packet in a communication protocol. One component may treat this as an uninterpreted collection of bits to be transmitted whereas another component may impose a structure on the packet with different fields indicating addresses, control, and checksum.

The following is a simple example of an interface. Consider a device that can be reserved by manipulating two signals (could be bits in a register, lines on a bus, variables, or wires): `request` and `grant`. Use of the device follows a simple four-phase protocol where a request is followed by a grant, after which the device can be used. When the use has finished, the request is removed and this is followed by the device removing the grant. The interface consists of the two signals plus the four-phase protocol. In Java the interface could be described as follows:

```
class device{
  private boolean request;
  private boolean grant;
}
```

In addition to the syntactical information specifying the number, names and types of the signals the interface defines a protocol requiring the pair `(request, grant)` to change as follows:

```
(false, false) -> (true, false) -> (true, true)
-> (false, true) -> (false, false) -> ...
```

The device is only assumed to work properly if all use of it follows this protocol. It is therefore an integral part of the interface that should be part of the written documentation.

In a systems design components from very different technologies are put together, and the interface description should be able to bridge the gaps between a range of technologies such as software (for a general purpose computer), software (for a specialized controller), various programmable hardware technologies such as FPGA, special purpose dedicated processors, synthesizable circuitry and hand-crafted ASICS.

*Work supported by the Danish Technical Research Council, project Codesign. E-mail and WWW address of the author: `jst@it.dtu.dk`, `http://www.it.dtu.dk/~jst`

0-8186-7996-4/97 $10.00

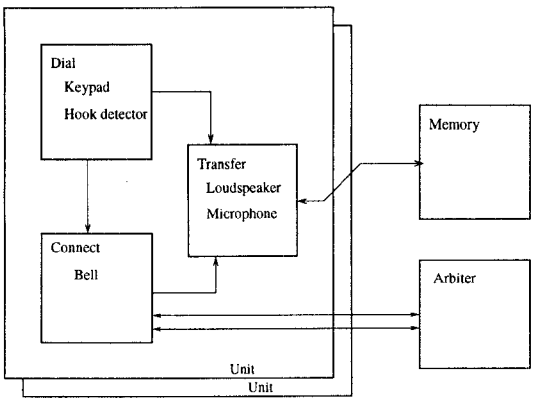

Figure 1: Overview of telephone switch

3 Teaching interface design

One goal of the experimental systems design course mentioned in this paper is to teach the students the importance of interface design. Designing and maintaining an interface require a delicate balance between specialization that can be exploited in the implementation of the module and generalization that makes the component more usable in a range of environments. A typical mistake made by many engineering students is to exploit some clever optimization based on implicit assumptions restricting the interface. By focusing on the importance of interface design and by insisting on consistency checks, such optimization can only be done when they do not violate the assumptions explicitly stated in the interface specification.

3.1 An example: The Telephone Switch

To illustrate the approach and to allow students to get some hands-on experience they are asked to design a simplified digital telephone switch (a simplified version of the Tigerswitch [1]). Fig. 1 gives an overview of the switch that can handle the switching of a number of phones. There is a separate unit for handling each phone, this unit has physical parts like the connector for plugging in the telephone and abstract parts for doing the necessary computation. The switch has an arbiter that reserves the buffers needed for communication and resolves conflicts like two phones calling the same receiver. Finally, the memory contains the buffers needed for exchanging data. As illustrated in Fig. 1, the switch is divided into five kinds of components (Dial, Connect, Transfer, Memory, and Arbiter).

Although the switch is quite simple it has a number of non-trivial interfaces, for example, between the dial and connect components. Section 3.3 describes techniques for documenting an interface that enables the designer to verify that the interface is interpreted consistently in different components. Type checking is a first step in that direction.

3.2 A Web based design environment

The design of the telephone exchange is done by a group of students starting out with an informal verbal description like the one given above. The students are divided into groups each designing a separate part of the exchange. Then an initial interface model is developed by each group and made available to all other participants via a set of Web pages. This set-up has several interesting properties:

- The students work independently of each other. All documentation is available to everybody at all times. This is a tremendous strength in an environment where participants work on several other tasks simultaneously (in case of university students, it is other courses, but in an industrial environment it would be other projects), and where work is done at all times (day and night). Hence, the need to physically meet to exchange information is greatly reduced.

- There is no need to separate documentation documents from the actual design documents. They can be one and the same set of files made visible to others via the Web. By doing this, documentation is always accurate and up to date.

- Ambiguities and misinterpretations are uncovered by insisting on written documentation as opposed to informal communication. During the course students are taught techniques for specifying interfaces and protocols (such as the four-phase protocol mentioned above).

The importance of interfaces and the penalties paid by careless treatment of interfaces become painfully clear in most large systems development projects. However, it can be difficult to illustrate such problems on smaller examples of a size that can be handled in the limited time-frame available in a university course. However, the Web based set-up for distributed and off-line design of the telephone switch illustrates the problems quite well.

3.3 Techniques for checking interfaces

This section briefly sketches a few verification techniques and tools for checking interface consistency. The interface of a component is *the externally visible behavior* such as the syntax and semantic interpretation of its communication with the environment.

An interface model of a component is a rigorous but possibly partial specification of a components interface. By allowing an interface to be partially specified, one leaves it to the designer to decide what should be rigorously defined in the interface model. The rigor makes it possible to decide whether different components have a consistent view of their common interface.

A simple form of interface checking is already routine in most systems design projects using some form of high level programming language for describing components. It is common practice to do type-checking across separately compiled components (the C header file mechanism is an example which makes it possible to do a simple check of interface consistency). However, the idea of type checking common interfaces can be taken much further [2]. Most modern high-level languages have a sophisticated module concept for handling a hierarchical library, the package concept in VHDL and the interface/class constructs in Java are good examples.

Although these module concepts are useful they only represent a small step towards what is feasible in terms of checking interface consistency. Traditional compilers make syntactical checks ensuring that the number and type of parameters are consistent. However, it is also important to avoid semantic inconsistencies such as one module assuming one protocol, for example, that a high value of a particular value means "go", while another module assumes a different protocol, e.g., that a high value means stop. The next section gives a brief description of how to do such checks.

3.3.1 Interface consistency

This section illustrates a technique for rigorously checking that different modules have a consistent view of their common interface.

As an illustration consider two modules X and Y with a common interface. Assume that I_X is a predicate that is satisfies by all the manipulations of the common interface done by module X. Similarly, I_Y is a predicate characterizing the interface manipulations done by Y. In case of the simple arbiter discussed in section 2 such an interface predicate could be $NOT\,(grant_X\ AND\ grant_Y)$. In this simple example both modules have the same predicate so they are obviously consistent. However, in general the predicates can be different which raises the possibility of inconsistency.

In addition to manipulations of the interface, a module may do local modifications. Assume that P_X and P_Y are predicates characterizing these. This means that all changes in module X must satisfy both: $P_X\ AND\ I_X$ (and similarly for Y). If $P_X\ AND\ I_X$ is strong enough to conclude that I_Y holds, and similarly if $P_Y\ AND\ I_Y$ is strong enough to conclude that I_X holds, then we may conclude that neither module violates the others requirements on the interface. Hence, interface verification consists of showing two implications such as:

$$P_X\ AND\ I_X \Rightarrow I_Y$$
$$P_Y\ AND\ I_Y \Rightarrow I_X$$

This sufficient condition is discussed in further detail in [3]. The Web based systems design course mentioned several times in this paper uses a suite of prototype tools for specifying and checking interface predicates.

The expressiveness of the notation used for the predicates is a key issue. The example above used simple predicates; there are many much more expressive proposals allowing interface predicates to include temporal properties. However, the more general the notation the more difficult it becomes to master it and to construct tools for automatically checking interface consistency.

4 Conclusion

The Web based systems design environment and the prototype interface verification tools sketched in this paper form the basis of an experimental systems design course. The students following the course realize the importance of interface design and consistency checking within a limited time frame and using (relatively) simple examples and tools.

References

[1] Wayne Wolf et. al. Tigerswitch: A case study in embedded computing system design. In *Proceedings from Codes/CASHE '94*, pages 89–96. IEEE Computer Society Press, September 1994.

[2] John V. Guttag, James J. Horning with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science, 1993.

[3] Jørgen Staunstrup and Niels Mellergaard. Localized verification of modular designs. *Formal Methods in System Design*, 6(3):295–320, 1995.