Interface Consistency

Jørgen Staunstrup
Department of Information Technology,
Technical University of Denmark,
DK-2800 Lyngby, Denmark

tel: +45 4525 3740 fax: +45 4593 0074

Email: jst@it.dtu.dk http://www.it.dtu.dk/~jst

Abstract

This paper proposes that *Interface Consistency* is an important issue for the development of modular designs. By providing a precise specification of component interfaces it becomes possible to check that separately developed components use a common interface in a coherent matter thus avoiding a very significant source of design errors. A wide range of interface specifications are possible, the simplest form is a syntactical check of parameter types. However, today it is possible to do more sophisticated forms involving semantic checks.

Keywords: Interfaces, consistency checking, formal verification, modular design, design languages.

1 Background

This paper stresses the importance of *interface design and verification* in the development of large modular designs. The primary reason for separating a design into distinct components is to enable the designer(s) to separate their concerns and work on one component at a time. These components need to communicate and coordinate their computations, the mechanism for doing this is called their *interface*, e.g., parameters to procedures and methods.

The separation of a design into distinct components creates the possibility that an interface is not treated consistently in all components communicating through the interface leading to errors and delays in design and development projects. When working on one component, the designer might treat the signaling on a communication line in one way and later when working on another component the signaling is treated differently. This is a very common source of errors in particular in large designs done by a large group of designers. What often happens is that the designers initially have an informal agreement about the conventions for using the interface. Later in the design process the details of this agreement are then interpreted differently in different components leading to an inconsistency. Interface verification consists of verifying that there are no such inconsistencies.

2 Position

The interface of a component is *the externally visible behavior* such as the syntax and semantic interpretation of its communication with the environment. Components can be either software modules, physical hardware devices or a combination of the two. Components can be newly created for a specific purpose, but often they are general purpose off-the-shelf components. In both cases it is important that the interface to the module is simple and well documented. 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.

2.1 Interface Consistency

This section illustrates a few verification techniques for rigorously checking that different modules have a consistent view of their common interface. A prerequisite for any kind of checking is to have a specification to check against. An interface specification of a component is a rigorous but possibly partial definition 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 specification. The rigor makes it possible to decide whether different components have a consistent view of their common interface.

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.

As an illustration consider two components X and Y with a common interface. Assume that I_X is a predicate that is satisfied 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 $\neg\,(grant_X \wedge grant_Y)$. In this simple example both components 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: $P_X \wedge\ I_X$ (and similarly for Y). If $P_X \wedge\ I_X$ is strong enough to conclude that I_Y holds, and similarly if $P_Y \wedge\ I_Y$ 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:

$$P_X \wedge I_X \Rightarrow I_Y$$

$$P_Y \wedge I_Y \Rightarrow I_X$$

This sufficient condition is discussed in further detail in [SM95]. Note that this is a weaker condition than requiring equivalence (or equality), hence the term *interface consistency*.

The expressiveness of the notation used for the predicates is a key issue. In the condition stated above, it was not explicitly stated how to write the predicates characterizing an interface, e.g., I_X. It could be a very restricted notation such as predicate logic with terms consisting only of program variables and constants or one could imagine higher-order logics allowing powerful quantifications. Another interesting possibility is to allow interface predicates to include temporal properties [PG97, RS94]. It is, however, not clear that notations which are very expressive from a theoretical viewpoint are also the most useful in practice. It is my experience that limited notations are often the ones giving the best results in practice, both because they are easier to master and because they allow more efficient tools to be constructed.

3 Comparison

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 [DET84, GwSGJ$^+$93]. The LCLint tool [DET84] is a nice example of how extended checking can be useful for revealing bugs before going to full scale testing.

Most modern high-level languages have a sophisticated module concept for handling a hierarchical library, the package concept in VHDL [VHD88] and the interface/class constructs in Java [AG96] 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. By documenting essential properties of an interface in formal notation, such as the interface predicates defined above, it becomes possible to exhaustively check interface consistency as described in the position section (and the references mentioned in that section).

In my view there are a number of interesting issues that need further work including:

1. *Experimentation* with the expressive power of the notation used for documenting interfaces. It is not clear that notations which are very expressive from a theoretical viewpoint are also the most useful in practice. An example is LCLint [DET84] which is a tool that practioners find very useful even if it is used in a very limited way.

2. *Timing* is a key issue in many interfaces, and it would of course be very useful to check timing properties of interfaces. There is work on formalizing timing diagrams e.g. in [PG97, CHR91], but I do not think that the final words on this issue have been said.

3. *Tool support* is important for any approach that wants to gain practical acceptance. Tremendous progress has been made over the past few years in model-checking tools, see e.g., [K.L93]. However, it is not clear that such finite state models are adequate for the more open ended formalism needed to document interfaces. Personally, I have experimented with tools based on general purpose theorem proves which makes it possible to check interfaces including infinite domains like integers (without bounds) and arrays without fixed index ranges [SM95].

References

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley Publishing Company, 1996.

[CHR91] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Proc. Letters*, 40(5), December 1991.

[DET84] Jim Horning D. Evans, John Guttag and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*, 1984.

[GwSGJ+93] 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.

[K.L93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[PG97] E. Cerny P. Girodias. Interface timing verification with delay correlation using constraint logic programming. In *Proceedings of ED & TC'97*, Paris, March 1997.

[RS94] Anders P. Ravn and Jørgen Staunstrup. Interface models. In *Proceedings from Codes/CASHE '94*, pages 157–164. IEEE Computer Society Press, September 1994.

[SM95] Jørgen Staunstrup and Niels Mellergaard. Localized verification of modular designs. *Formal Methods in System Design*, 6(3):295–320, 1995.

[VHD88] IEEE, New York. *VHDL Language Reference Manual*, std 1076-1987 edition, 1988.

4 Biography

Jørgen Staunstrup is Full Professor in Computer Science at the Technical University of Denmark, Lyngby. His research interest are formal verification, design of integrated circuits and design languages. The book "A Formal Approach to Hardware Design" published by Kluwer in 1994 presents a formally based approach to the design of integrated circuits. He has previously done research in programming languages and multiprocessor algorithms. The results of this work has been exploited by Danish electronics companies. He has taught university courses on programming, algorithms and datastructures, formal verification, specification, operating systems and VLSI design.

Jørgen Staunstrup has a Ph. D. in Computer Science from University of Southern California, Los Angeles in 1978 and a Dr. techn. degree from The Technical University of Denmark in 1995. He has had visiting appointments at the University of Washington, Seattle, Massachusetts Institute of Technology, Cambridge and INRIA, Paris.

He is a member of IFIP WG 10.5, IFIP TC 10, and vicechairman of the Danish Technical Research Council.