6.826 Principles of Computer Systems **Fall Semester, 1991**

Lecture 1: September 14, 1992

Lecturer: Bill Weihl *Scribe: Fred Chong*[1]

1 Course Outline

This course is a graduate level introduction to principles of computer systems. It will focus on concurrent and distributed systems. It is not based on a series of case studies. Instead we will attempt to examine systems in a common framework of rigorous specification and reasoning.

The general theme of the course is set around these topics:

- **Specifications** — which are used to abstract the visible behavior of a system from the details of its implementation.
- **Reasoning techniques** — aimed at proving the correctness of a system.
- **Performance** — including issues in design tradeoffs, and ways to enhance performance.

We shall discuss these issues in the context of some specific areas and techniques listed below :

- **Performance optimization**, such as caching as a method to improve performance through the use of a *faster*, *smaller*, and *nearer* memory to store frequently used data.
- **Fault-tolerance**, which is about building systems that keep working in the wake of faults.
- **Concurrency**, implying that several processes are allowed to run in parallel and interact with each other. We will discuss problems involved in specifying and reasoning about concurrency. This course does *not* cover a study of parallel algorithms or applications for solving *specific* problems.
- **Distributed systems**, which are spread across multiple nodes connected by a network. This introduces some new issues like communication delays and independent failure of the nodes.
- **Transactions**, as a mechanism for building fault-tolerant, distributed, and concurrent systems.
- **Naming**, the issues involved in mapping names to resources and the naming problems introduced by distributed systems.
- **Security**, involving matters like access-control and authentication.

[1]These notes based on those of Umesh Maheshwari of September 11, 1991.

2 Principles of Computer System Design

A list of principles useful in the design of computer systems could conceivably be very long. As noted earlier, we shall focus on three factors: abstractions, reasoning techniques, and tradeoffs between performance, functionality and complexity.

Abstraction refers to the process of using specifications to separate the functionality of a system from the details of its implementation. In other words, specifications tell *what* a system should do, without trying to dictate *how* it should be done. As an example, the specification for a $SORT$ routine may state merely that it permutes the input sequence of numbers such that the output sequence has them in increasing order. The particular sorting algorithm used to achieve this is irrelevant. Actually, the distinction between what is and is not a part of a specification is subjective, and depends upon the which aspects of the system's behavior its users find interesting. For instance, if the $SORT$ routine is required to run in $O(n \log n)$ time, then this fact would appear in the specifications, and less efficient implementations like $InsertionSort$ would be ruled out.

The use of abstractions in design has some obvious advantages :

- It provides modularity and modifiability. That is, the implementation of a module can be replaced without changing its specification, and therefore, other modules using it remain unaffected.
- A related point is that it allows the designer to avoid pre-mature optimization. For example, one could employ an inefficient but simple-and-sure implementation to begin with, and worry about performance only after the integration of the system is complete.
- It is a good way to document functionality.

Specifications can be viewed as contracts between users and implementors. The users must rely *only* on the properties in the specification. The implementors must provide those properties. An implementation may have certain fortuitous unspecified features. However, users should not rely upon them. The implementors are free to change such features in future versions of the system.

For an example of a specification, let us attempt to list the essential features of a wristwatch:

- The displayed time must be accurate to some small delta.
- User interface – the display and input mechanisms must be of a certain format.
- The energy source must have a reasonable lifetime.
- The watch must be waterproof to a certain depth
- Aesthetics – the watch must be under a certain weight, the strap of a certain type.
- The price must be under a certain amount.

Figure 1: Performance/functionality tradeoff in a Distributed System.

However, listing generic features of a wristwatch is a bit unfair. The specification really depends upon user needs.

The behavior of modules is usually well-understood in a sequential world, but has more complicated meaning for concurrent and distributed systems. Specifications for a concurrent system are often *weaker* than those for a sequential system. This results from the non-determinacy allowed by the presence of concurrency. Forcing determinism in such a system may result in degradation of performance.

As an illustration, consider a network of file servers and clients (Fig 1). If *Client A* updates a file before *Client B* reads it, the most straightforward semantics would guarantee that the modifications are reflected in the data read by *Client B*. However, performance would improve tremendously if *Client B* could use a cached (potentially old) copy of the file. In this scenario, modifications to the file are propagated to *Client B* shortly after they occur. The extra performance requires that the semantics be relaxed; thus, there are tradeoffs between good performance and the desirable functionality with simpler specifications.

3 Writing Specifications

In this course, both specifications and implementations will be written using one language — *Spec*. Spec is based on Edsger Dijkstra's *Guarded Commands*[2], extended by Greg Nelson[3], and augmented with constructs for concurrency and atomicity by Butler Lampson.[4]

Using only one language allows successive refinement of a specification into a full-fledged implementation. On the other hand, this has some pitfalls of its own. It is easy for one to lapse into using the implementation style while coding the specifications, which makes the specifications *overconstrained*. Indeed, an implementation *is* a plausible specification. Conversely, one might be tempted to use some of the succinct constructs provided for specifications while writing the implementation, which may not translate into realistic or efficient code. Thus, we need to exercise some 'self-restraint' in using Spec for the two different purposes.

We now turn to what the specification for a procedure might look like. It could conceivably be a function mapping the input arguments and the initial state to the results returned and the final state. Here, the results encapsulate both the values returned normally and the exceptions raised. However, the use of a *function* for the specification fails to capture the non-deterministic behavior of some procedures. That is, for the same input and initial state, a number of output values or state may be acceptable. Some examples of such procedures are:

- $SQRT(x)$ which might be allowed to return any value within $\sqrt{x} \pm \epsilon$ for want of better precision.
- $SEARCH(item : T, s : SEQ[T])$ which returns any index j in s, such that $s[j] = item$, or raises a $NotFound$ exception if none exists.
- The storage allocator, which might return any of the several available free blocks on request for a chunk of memory.

Thus the specification for a procedure should be expressed as a *relation* between the input arguments plus the initial state, and the results plus the final state — this allows one-to-many mapping. In fact, even a particular *implementation* of the procedure may be nondeterministic (e.g., if it involves concurrency), and then it too must be denoted by a relation rather than a function.

Suppose that the relation dictated by the specification is R_S and that realized by the implementation is R_I. The requirement that the implementation should satisfy the specification translates as

$$R_I \subseteq R_S$$

In words, all possible outcomes of invoking the implementation on a given set of arguments and initial state must be included in the outcomes allowed by the specification. It is possible that R_I is actually a function while R_S is not. For instance, if an implementation of the $SEARCH$ routine

[2]See Dijkstra's paper called *Guarded Commands, Nondeterminacy and Formal Derivation of Programs* in the Aug'75 issue of *Communications of the ACM*.

[3]See Nelson's *A Generalization of Dijkstra's Calculus*, ACM Transactions on Programming Languages and Systems, 11(4):517–561, October 1989

[4]Neither of these references will help students learn Spec; they are provided only for historical interest.

discussed above always returns the first location (if any) of the item in the sequence, then the implementation is indeed deterministic. Implicit in the subset relation is the requirement that the domain of R_I be the same as the domain of R_S, otherwise the implementation ignores some legal inputs.

Before winding-up, we take a look at issues of *termination*. It is possible for a procedure to not terminate at all for some inputs. We can denote this case by a special state $\perp$, pronounced *bottom*. An implementation is said to be *partially correct* if it satisfies the specification whenever it terminates. It is said to be *totally correct* if it is partially correct and always terminates.