The Book Review Column[1]

by William Gasarch
Department of Computer Science
University of Maryland at College Park
College Park, MD, 20742
email: `gasarch@cs.umd.edu`

In this column we review the following books.

1. **Concurrent and Real-time Systems: The CSP Approach** by Steve Schneider. Reviewed by Sabina Petride. This book is about Communicating Sequential Processes (CSP) which is a model of concurrent processes. It is used to verify protocols. and mode

2. **Introduction to Languages, Machines and Logic: Computable Languages, Abstract Machines and Formal Logic** by Alan P. Parkes. Reviewed by Robert McNaughton. This is a textbook for a course on Automata theory and related topics.

3. **Algorithm Design: Foundations, Analysis and Internet Examples** by Michael T. Goodrich and Roberto Tamassia. Reviewed by Pavol Navrat. This is a texbook for a course in algorithms.

4. **Theory of Semi-Feasible Algorithms** by Lane Hemaspaandra and Leen Torenvliet. Review by Lance Fortnow. This is a monograph on the topic of P-selective sets.

Books I want Reviewed

If you want a FREE copy of one of these books in exchange for a review, then email me at gasarchcs.umd.edu Reviews need to be in LaTeX, LaTeX2e, or Plaintext.

Books on Algorithms

1. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica* by Pemmaraju and Skiena.

2. *Algorithms: Design Techniques and Analysis* by Alsuwaiyel.

3. *Foundations of Algorithms Using Java Psudeocode* by Neapolitan and Naimipour.

4. *Computational Techniques of the Simplex Method* by Maros.

5. *Immunocomputing: Principles and Appliations* by Tarakanov, Skormin, Sokolova.

6. *Data Structures, Near Neighbor Searches, and Methodoolgy: Fifth and Sixth DIMACS Implementation Challenges* Edited by Michael Goldwasser, David Johnson, Catherine McGeoch.

Books on Cryptography

1. *Data Privacy and Security* by David Salomon.

2. *Elliptic Curves: Number Theory and Cryptography* by Larry Washington.

3. *Block Error-Correcting Codes: A Computational Primer* by Xambo-Descamps.

[1]© William Gasarch, 2004.

Misc Books

1. *Turing (A novel about computation) by Christos Papadimitriou.* (This is really a novel!)
2. *Dynamic Reconfiguration: Architectures and Algorithms* by Vaidyanathan and Trahan.
3. *Semantic integration of heterogeneous software specifications* by Gro*ß*e-Rhode.
4. *Finite Automata* by Lawson.
5. *Ordered Sets: An Introduction* by Schroder.
6. *Combinatorial Designs: Constructions and Analysis* by Stinson.

Review[2] of
Concurrent and Real-time Systems: The CSP Approach
Authur: Steve Schneider
526 pages, $70.00, 1999, Paperback
Publisher: Wiley
Review by Sabina Petride

1 Overview

1.1 Sequential processes

Much has been done to formally talk about processes in the last decades, especially in the field of process algebras. In [Hoa85] A. Hoare laid the foundations of Communicating Sequential Processes (CSP) formalism, while in [Mil89] Milner proposed the slightly different approach of Calculus of Communicating Systems (CCS). Both process algebras have been extensively used to model and verify concurrent systems, and recent publications prove the richness of the formalisms ([Ros97], [Mil99], [Fok00]). *Concurrent and Real-time Systems. The CSP Approach* was published in 1999 and since then has been one of the most referenced books on CSP.

Let us start with the basics. CSP provides a language for writing processes that perform local or external events and communicate with each other. It is very important to understand from the very beginning the view that CSP has on events/actions: there are *external* events from a set Σ and only one *internal* event named τ, which means that CSP focuses on the external behavior of processes and regards all possible internal events as identical.

For the beginning, we consider the language for sequential processes, without taking communication into account.

The simplest process is $STOP$: it does not perform any action; the next simplest process is $SKIP$, which can do nothing except performing the special termination event $\surd$.[3] $a \longrightarrow P$ (a then P) performs event a and then behaves as the process P. A process that can engage in any of the events in set A and, after choosing some $a \in A$, behaves as $P(a)$ is called *menu* (*prefix*) choice and is written as $x : A \longrightarrow P(x)$.

The CSP language explictly uses channels and associates to each channel c a type T of values that can be sent or received along c; $c.T \overset{def}{=} \{c.t \mid t \in T\}$ represents the set of events associated with c. The process $c!v \longrightarrow P$ outputs value v on channel c and then behaves as P, while process

[2]©2004 Sabina Petride

[3]If A is a set of events, then $A^{\surd} = A \cup \{\surd\}$.

$c?x \longrightarrow P(x)$ is waiting for some value v of type T to be sent on channel c and then behaves as $P(v)$.

Similar to the prefix choice construct, there are built in terms to account for choice among processes. If the choice between processes P_1 and P_2 is resolved by the performance of the first event, in favor of the process that performs it, then the choice is called *external* and the CSP term is written as $P \Box P$. If the choice between P_1 and P_2 is done internally, and the environment has no control on it, then the choice is called *nondeterministic* (*internal*) and the CSP term is written as $P \sqcap P$. If we add (mutual) recursion by allowing definitions of the form $N = P$ where P includes name N, then we have the core CSP language for sequential processes:

$$\begin{array}{lll} P & ::= & STOP \mid SKIP \mid a \longrightarrow P \mid x : A \longrightarrow P(x) \\ & & c!v \longrightarrow P \mid c?x : T \longrightarrow P(x) \\ & & P \Box P \mid P \sqcap P \end{array}$$

The above informal explanations of the behavior of CSP sequential processes is made rigorous by the operational semantics:[4]

$$\frac{}{(a\longrightarrow P)\xrightarrow{a}P} \qquad \frac{}{(x{:}A\longrightarrow P(x))\xrightarrow{a}P(a)}[a \in A]$$

$$\frac{}{SKIP\xrightarrow{\checkmark}STOP} \qquad \frac{P\xrightarrow{\mu}P'}{N\xrightarrow{\mu}P'}[N = P]$$

$$\frac{}{(c!v\longrightarrow P)\xrightarrow{c.v}P} \qquad \frac{}{(c?x\longrightarrow P)\xrightarrow{c.v}P(v)}[v \in T]$$

$$\frac{P_1\xrightarrow{a}P_1'}{P_1\Box P_2\xrightarrow{a}P_1'}[a \in \Sigma^{\checkmark}] \qquad \frac{P_2\xrightarrow{a}P_2'}{P_1\Box P_2\xrightarrow{a}P_2'}[a \in \Sigma^{\checkmark}]$$

$$\frac{P_1\xrightarrow{\tau}P_1'}{P_1\Box P_2\xrightarrow{\tau}P_1'\Box P_2} \qquad \frac{P_2\xrightarrow{\tau}P_2'}{P_1\Box P_2\xrightarrow{\tau}P_1\Box P_2'}$$

$$\frac{}{P_1\sqcap P_2\xrightarrow{\tau}P_1} \qquad \frac{}{P_1\sqcap P_2\xrightarrow{\tau}P_2}$$

Control flow is modeled by sequential composition $(P_1; P_2)$ and interrupt $(P_1 \triangle P_2)$. $P_1; P_2$ behaves as P_1 followed by P_2 if P_1 terminates. $P_1 \triangle P_2$ represents a process that mimics P_1, but at any point it can interrupt P_1 and behave as P_2; once the control has been removed from P_1, it cannot be given back to P_1; P_2 may perform internal actions while P_1 has control. To account for these behaviors, the operational semantics is extended with the following rules:

$$\frac{P_1\xrightarrow{\mu}P_1'}{P_1;P_2\xrightarrow{\mu}P_1';P_2} \qquad \frac{P_1\xrightarrow{\checkmark}P_1'}{P_1;P_2\xrightarrow{\tau}P_2}$$

and

[4]The operational semantics is a set of rules that specify the possible transitions terms can engage in. A rule has a (possibly empty) set of conditions (written above the horizontal line), and a set of conclusions; additional constraints on the rule are placed in brackets, at the right side of the line.

$$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \triangle P_2 \xrightarrow{\mu} P_1' \triangle P_2}[\mu \neq \surd] \quad \frac{P_1 \xrightarrow{\surd} P_1'}{P_1 \triangle P_2 \xrightarrow{\surd} P_1'} \quad \frac{P_2 \xrightarrow{\tau} P_2'}{P_1 \triangle P_2 \xrightarrow{\tau} P_1 \triangle P_2'} \quad \frac{P_2 \xrightarrow{a} P_2'}{P_1 \triangle P_2 \xrightarrow{a} P_2'}$$

1.2 Communicating Sequential Processes. Concurrency

It is time now to focus on how the CSP language models concurrency and communication. One additional notion is required: the *interface* of a process P is defined as all external events that P may perform. CSP processes can execute concurrently by synchronizing on common events: any event that appears in the interface of both processes must involve both processes, whenever it occurs. The synchronization mechanism resembles handshake synchronization, since it occurs only when all participants engage in it simultaneously.

Consider the case when we want to put two processes P_1 and P_2 in parallel. If P_1's interface is A and P_2's interface is B, then P_1 in parallel with P_2 must synchronize on $A \cap B$, while allowing P_1 to execute any event not in B that it can independently perform, and similar for P_2. We write the resulting process $P_1 \;{}_{\mathrm{A}}\|_{\mathrm{B}}\; P_2$ and name it P_1 *parallel on A, B* P_2. The operational semantics is extended with:

$$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \;{}_{\mathrm{A}}\|_{\mathrm{B}}\; P_2 \xrightarrow{\mu} P_1' \;{}_{\mathrm{A}}\|_{\mathrm{B}}\; P_2}[\mu \in (A \cup \{\tau\}) \backslash B] \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_2 \;{}_{\mathrm{B}}\|_{\mathrm{A}}\; P_1 \xrightarrow{\mu} P_2 \;{}_{\mathrm{B}}\|_{\mathrm{A}}\; P_1'}[\mu \in (A \cup \{\tau\}) \backslash B]$$

$$\frac{P_1 \xrightarrow{a} P_1', P_2 \xrightarrow{a} P_2'}{P_1 \;{}_{\mathrm{A}}\|_{\mathrm{B}}\; P_2 \xrightarrow{a} P_1' \;{}_{\mathrm{A}}\|_{\mathrm{B}}\; P_2'}[a \in A^{\surd} \cap B^{\surd}] \quad \frac{P_1 \xrightarrow{a} P_1', P_2 \xrightarrow{a} P_2'}{P_2 \;{}_{\mathrm{B}}\|_{\mathrm{A}}\; P_1 \xrightarrow{a} P_2' \;{}_{\mathrm{B}}\|_{\mathrm{A}}\; P_1'}[a \in A^{\surd} \cap B^{\surd}]$$

Similarly, P_1 *interface A parallel* P_2 is written $P_1 \underset{A}{\|} P_2$ and its operational semantics is simply

$$\frac{P_1 \xrightarrow{a} P_1', P_2 \xrightarrow{a} P_2'}{P_1 \underset{A}{\|} P_2 \xrightarrow{a} P_1' \underset{A}{\|} P_2'}[a \in A^{\surd}] \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \underset{A}{\|} P_2 \xrightarrow{\mu} P_1' \underset{A}{\|} P_2}[\mu \notin A^{\surd}] \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_2 \underset{A}{\|} P_1 \xrightarrow{\mu} P_2 \underset{A}{\|} P_1'}[\mu \notin A^{\surd}].$$

If, given two processes P_1 and P_2, we want to model their independent execution in parallel, then we use the CSP term $P_1 \,|||\, P_2$ (P_1 *interleave* P_2) with the operational semantics

$$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 ||| P_2 \xrightarrow{\mu} P_1' ||| P_2}[\mu \neq \surd] \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_2 ||| P_1 \xrightarrow{\mu} P_2 ||| P_1'}[\mu \neq \surd] \quad \frac{P_1 \xrightarrow{\surd} P_1', P_2 \xrightarrow{\surd} P_2'}{P_1 ||| P_2 \xrightarrow{\surd} P_1' ||| P_2'}.$$

With communicating processes comes the need to rename some events in a process interface or to specify events we do not care about. Renaming in CSP is done by using functions of the form $f : \Sigma^{\surd} \longrightarrow \Sigma^{\surd}$ with $f(a) = \surd \Leftrightarrow a = \surd$ and writing $f(P)$ for the process with the following behavior:

$$\frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')} \quad \frac{P \xrightarrow{\tau} P'}{f(P) \xrightarrow{\tau} f(P')}.$$

For a process P we can take some set of events A from the interface and *hide* them (i.e. make them internal); the resulting process is $P \backslash A$ and the operational semantics is extended with

$$\frac{P \xrightarrow{a} P'}{P\backslash A \xrightarrow{\tau} P'\backslash A}[a \in A] \quad \frac{P \xrightarrow{\mu} P'}{P\backslash A \xrightarrow{\mu} P'\backslash A}[\mu \notin A].$$

We have now covered the complete untimed CSP language:

$$P \;\; ::= \;\; ... \mid P_1 \;{}_{\mathrm{A}}\|_{\mathrm{B}}\; P_2 \mid P_1 \mid\mid\mid P_2 \mid P_1 \underset{A}{\|} P_2 \mid f(P) \mid P\backslash A.$$

1.3 Analyzing processes: semantics, specification, and verification

We have all the necessary sophistication to understand the famous problem of dining philosophers (see page 79).[5] There are 5 philosophers numbered from 0 to 4 sitting at a circular table, and in between each of them there is a chopstick: chopstick i is in between philosophers i and $(i-1)\, mod 5$. To each philosopher and chopstick is associated a process expressed in CSP named $PHIL_i$, $CHOP_i$ respectively; the following channels are considered: *enter*, *eat*, *leave*, *pick*,*put*. To understand the behavior of each process some events have to be explained:

- *enter.i*: philosopher i enters the dining room
- *eat.i*: philosopher i eats
- *leave.i*: philosopher i leaves the dining room
- *pick.i.i* philosopher i picks the right-hand chopstick
- *pick.i.*$((i+1)\, mod\, 5)$: philosopher i picks the left-hand chopstick
- *put.i.i*: philosopher i replaces the right-hand chopstick
- *put.i.*$((i+1)\, mod\, 5)$: philosopher i replaces left-hand chopstick

Philosopher i repeats the following behavior: enters the room and then either picks the chopsticks (right first or left first), eats, replaces the chopsticks in the order he picked them, and finally leaves. This means that the corresponding CSP process is:

$$\begin{aligned} PHIL_i \;\; = \;\; & enter.i \longrightarrow \\ & ((pick.i.i \longrightarrow pick.i.((i+1)\, mod\, 5) \longrightarrow eat.i \longrightarrow \\ & \quad put.i.i \longrightarrow put.i.((i+1)\, mod\, 5) \longrightarrow leave.i \longrightarrow PHIL_i) \\ & \Box \\ & \quad ((pick.i.((i+1)\, mod\, 5) \longrightarrow pick.i.i \longrightarrow eat.i \longrightarrow \\ & \quad \longrightarrow put.i.((i+1)\, mod\, 5) \longrightarrow put.i.i \longrightarrow leave.i \longrightarrow PHIL_i)) \end{aligned}$$

Since philosophers do not synchronize on any events, their combination may be written as

$$PHILS = PHIL_1 \mid\mid\mid PHIL_2 \mid\mid\mid PHIL_3 \mid\mid\mid PHIL_4.$$

Each chopstick is repeatedly picked and then replaced by some of the two neighboring philosophers:

[5]an alternative modeling of this problem can be found in [Hoa85] in chapter 2.5

$$\begin{aligned} CHOP_j \;&= \; pick.j.j \longrightarrow put.j.j \longrightarrow CHOP_j \\ &\;\;\Box \\ &\;\; pick.((j-1)\,mod\,5).j \longrightarrow put.((j-1)\,mod\,5).j \longrightarrow CHOP_j \end{aligned}$$

The combination of all chopsticks is

$$CHOPSTICKS = CHOP_1 \,|||\, CHOP_2 \,|||\, CHOP_3 \,|||\, CHOP_4.$$

The system is thus the synchronized combination of philosophers and chopsticks:

$$SYS = PHILS \,||\, CHOPSTICKS.$$

How can we check that this system allows any philosopher who enters the room to eventually eat? We can write first a CSP term that accounts for the condition that some philosopher i that enters the room eventually eats:

$$SPEC = enter.i \longrightarrow eat.i \longrightarrow SPEC.$$

$SPEC$ is called the specification term and now we have to check that each behavior of SYS is a possible behavior of $SPEC$.

This is what happens when using CSP to solve problems: first we write a CSP term (SYS) for the system we want to model and then we write a CSP term ($SPEC$) that describes the specification, or the ideal system we have in mind. To check if SYS has the desired property, we have to check if any possible behavior of SYS is allowed by the specification $SPEC$. The only piece missing is what is understood by *behavior* of a process. This may seem a simple question, but the range of possible answers is quite large.

The behavior of a process is usually characterized by the sequences of events that the process performs. Such a sequence is called *trace* and for each particular CSP process P we can derive its traces ($traces(P)$) based on the operational semantics given above. To check that if philosopher i enters the room, he will eventually eat, we have to verify if

$$traces(SYS) \subseteq traces(SPEC).$$

If this is the case, then we say that $SPEC$ is *trace-refined* by SYS.

If the specification is given as predicate S on traces, then an implementation P is correct exactly when $S(tr)$ is true for all traces tr of P; we write this as $P\ sat\ S(tr)$. Based on the operational semantics of CSP processes, we can write a proof system for trace specifications. $P\ sat\ S(tr)$ is proved if it can be derived from the axioms and the rules of the proof system. As an example, a simple rule states that a process P satisfies the conjunction of two predicates on traces when P satisfies each of the predicates; this is formally written as

$$\frac{P\ sat\ S(tr),\, P\ sat\ T(tr)}{P\ sat\ (S \wedge T)(tr)}.$$

Similar techniques are used to study the behavior of a given process when interacting with specific environments. The desired environment is modeled as a CSP process T that is successful when performing the success event ω. As an example, in *may* testing, a process P passes the test T if, when placed in parallel with T, a success event is eventually performed: $P\ may\ T \equiv ((P \underset{\Sigma}{\|} T)\backslash\Sigma \overset{<\omega>}{\Longrightarrow})$.

Two process P_1 and P_2 are equivalent under *may* testing ($P_1 \equiv_{may} P_2$) if and only if

$$\forall T.(P_1\ may\ T \Leftrightarrow P_2\ may\ T).$$

Similarly, $P_1 \sqsubseteq_{may} P_2$ if and only if $\forall T.\neg(P_1\ may\ T) \Leftrightarrow \neg(P_2\ may\ T)$. *May* testing is nicely connected to trace refinement, since $P_1 \equiv_{may} P_2$ if and only if $traces(P_2) \subseteq traces(P_1)$.

More refined semantics for processes include stable failures and divergences, and for each possible model there are known verification techniques (see section 2 for more details).

1.4 Real-time Systems. The timed language

The full power of the CSP language relies in the elegant extension to account for timed processes. The first step to capture time is made by the *evolution relation*: we write $P \overset{d}{\leadsto} Q$, with $d > 0$ to stand for P becoming Q after d units of time, with no internal or external events performed at any point during this transition.

The operational semantics is given by:

$$\frac{}{(a \longrightarrow Q) \overset{d}{\leadsto} (a \longrightarrow Q)} \cdots \quad \frac{Q_1 \overset{d}{\leadsto} Q_1', Q_2 \overset{d}{\leadsto} Q_2'}{Q_1 \;{}_{A}\|_{B}\; Q_2 \overset{d}{\leadsto} Q_1' \;{}_{A}\|_{B}\; Q_2'} \quad \frac{Q_1 \overset{d}{\leadsto} Q_1', Q_2 \overset{d}{\leadsto} Q_2'}{Q_1 \mid\mid\mid Q_2 \overset{d}{\leadsto} Q_1' \mid\mid\mid Q_2'} \quad \frac{Q_1 \overset{d}{\leadsto} Q_1', Q_2 \overset{d}{\leadsto} Q_2'}{Q_1 \underset{A}{\|} Q_2 \overset{d}{\leadsto} Q_1' \underset{A}{\|} Q_2'}$$

[6]

$$\frac{Q \overset{d}{\leadsto} Q', \forall a \in A. \neg(Q \overset{a}{\longrightarrow})}{Q \setminus A \overset{d}{\leadsto} Q' \setminus A} \quad \frac{Q \overset{d}{\leadsto} Q'}{f(Q) \overset{d}{\leadsto} f(Q')} \quad \frac{Q_1 \overset{d}{\leadsto} Q_1', \neg(Q_1 \overset{\checkmark}{\longrightarrow})}{Q_1 ; Q_2 \overset{d}{\leadsto} Q_1' ; Q_2} \quad \frac{Q_1 \overset{d}{\leadsto} Q_1', Q_2 \overset{d}{\leadsto} Q_2'}{Q_1 \triangle Q_2 \overset{d}{\leadsto} Q_1' \triangle Q_2'}.$$

The event prefix process has to incorporate the passing of time; the operational semantics for timed event prefix is

$$\frac{}{(a@u \longrightarrow Q) \overset{a}{\longrightarrow} Q[0/u]} \quad \frac{}{(a@u \longrightarrow Q) \overset{a}{\leadsto} (a@u \longrightarrow Q[u+d/u])}.$$

Q_1 *timeout* d Q_2 ($Q_1 \overset{d}{\rhd} Q_2$) is the process that behaves as Q_1 if Q_1 performs some external event before d units of time, or as Q_2 otherwise:

$$\frac{Q_1 \overset{a}{\longrightarrow} Q_1'}{Q_1 \overset{d}{\rhd} Q_2 \overset{a}{\longrightarrow} Q_1'} \qquad \frac{Q_1 \overset{\tau}{\longrightarrow} Q_1'}{Q_1 \overset{d}{\rhd} Q_2 \overset{\tau}{\longrightarrow} Q_1' \overset{d}{\rhd} Q_2}[d > 0], \ \frac{}{Q_1 \overset{0}{\rhd} Q_2 \overset{\tau}{\longrightarrow} Q_2}$$

$$\frac{Q_1 \overset{d'}{\leadsto} Q_1'}{Q_1 \overset{d}{\rhd} Q_2 \overset{d'}{\rhd} Q_1 \overset{d-d'}{\leadsto} Q_2}[0 < d' \leq d] \quad \frac{}{Q_1 \overset{0}{\rhd} Q_2 \overset{\tau}{\longrightarrow} Q_2}.$$

As an example, the process that accounts for d units of time delay can be written as $WAIT\ d \equiv STOP \overset{d}{\rhd} S$

The timed CSP has a special construct for time interrupt: $Q_1 \triangle_d Q_2$ allows the execution of Q_1 for d units of time, but if by d units of time Q_1 did not finish, then Q_2 is executed.

$$\frac{Q_1 \overset{\mu}{\longrightarrow} Q_1'}{Q_1 \triangle_d Q_2 \overset{\mu}{\longrightarrow} Q_1' \triangle_d Q_2}[\mu \neq \surd] \quad \frac{Q_1 \overset{\checkmark}{\longrightarrow} Q_1'}{Q_1 \triangle_d Q_2 \overset{\checkmark}{\longrightarrow} Q_1'}$$

$$\frac{Q_1 \overset{d'}{\leadsto} Q_1'}{Q_1 \triangle_d Q_2 \overset{d'}{\leadsto} Q_1' \triangle_{d-d'} Q_2}[d' \leq d] \quad \frac{}{Q_1 \triangle_0 Q_2 \overset{\tau}{\longrightarrow} Q_2}.$$

The behavior of a process written in timed CSP can be expressed by timed observations, or alternatively by timed failures; similar in spirit with the approach for untimed CSP, there are

[6]The maximal progress requirement imposes that hidden events become urgent; also, evolution cannot occur when termination is possible.

methods for testing, writing timed specifications, doing verification, and characterizing timewise refinement.

2 Contents

In **Chapter 1** the reader is familiarized with the basic notions of process and event. The examples appeal to our common understanding of the world as a system with interacting components that evolve in time. Starting with the simplest process of all (STOP) and the *a then P* process, more complexity is added by explaining the concepts of internal/external choice and recursion. The formal notation builds the bridge from pictures and transition diagrams to operational semantics, with no apparent effort.

Chapter 2 focuses on processes executed concurrently and on the concept of synchronization. The notions of (indexed) alphabetized parallel combination, interleaving and deadlock are thoroughly explained with the aid of real-life (2-pump garage, multiple-line phone service), as well as more technical examples (chain of buffers passing on an item of data, network for array sorting, routing messages in a network).

The introductory part of book is completed in **chapter 3** with the discussions on abstraction mechanisms (hiding, event renaming and backward renaming), and control flow descriptions (sequential composition and interrupts). Each of these notions is justified by simple examples as print queues or chaining of multiple instances of the same process; the link with the first two chapters is strengthened by reinterpretations and extensions of previous examples. At the end of the chapter the reader has a good grasp on the core language of CSP and process execution modeled by operational semantics.

The second part of the book is concerned with semantic models for concurrent processes and techniques for specification and verification. The simplest semantics for concurrent processes is given by traces, as sequences of events which represent executions. **Chapter 4** gives the detailed definitions of traces for each type of CSP process, with a particular emphasis on recursion and the underlying fixed point theory. It also introduces a complementary approach to understanding processes in terms of their responses to other CSP processes (tests): two processes are equivalent under a certain class of tests if their responses to identical test processes are the same. It is proved that equivalence under *may* testing amounts to equality of trace sets; trace-refinement interpretations and the desired relation between implementation traces and specification traces are considered.

The importance of the trace model becomes more clear in **chapter 5** with the description of the proof system for trace specifications: requirements are given in terms of predicates on traces and the compositional nature of trace semantics for CSP is exploited to generate the set of proof rules with conclusion of the type $P\ sat\ S(tr)$ (CSP process P satisfies trace specification S). The section containing a detailed proof of the distributed sum problem makes clear the use of trace semantics and its corresponding proof system for verification of *safety properties*, e.g. properties of the form "something bad will not happen".

However, trace information is not always sufficient for identifying a process' responses when placed in different environments; as an example, $a \longrightarrow STOP \Box b \longrightarrow STOP$ and $a \longrightarrow STOP \sqcap b \longrightarrow STOP$ have the same traces, but an environment that communicates on event a succeeds when running concurrently with the external choice process, and still has an unpredictable behavior while interacting with the internal choice process. This justifies the need for more detailed observations of CSP processes. The *stable failure* semantics is concerned with the possible refusals of a process; the corresponding property(or process)-oriented specifications and proof systems for verification allow the analysis of nondeterminism and deadlock. The stable failures associated with the CSP processes make the subject of **chapter 6**, while the specifications and proof system for verification are presented in **chapter 7**. Additionally, if we test the system's *liveness properties* (properties of the type "something good will happen"), then we have to take into account possible divergent behaviors of processes; the *failure-divergences-infinite traces* (FDI) model gives CSP processes a more refined semantics, and its corresponding proof system for verification successfully addresses the liveness question. The model is presented in **chapter 8**. An excellent link with the previous semantics is established by focusing on the distributed sum example and gradually increasing the complexity of the model: the trace model for proving correctness (chapter 5), the stable failure semantics for showing liveness on inputs and outputs and deadlock-freedom for the network (chapter 7), and finally the FDI approach for proving divergence-freedom (chapter 8).

In **chapter 9** time is added to the framework. The timed CSP language and the operational semantics are thoroughly explained with the aid of numerous examples. Evolution transitions are given for the core CSP language, while the previous untimed transitions are preserved; new constructs are added to account for timeouts, minimal and maximal delays, timed interrupts, and timed event prefixes.

When using timed CSP it is of vital importance to fully understand the underlying assumptions and the corresponding constraints on processes evolutions (**chapter 10**). It is assumed that

- there is only one possible result that a process can reach simply by allowing time to pass (time determinism),
- the state a process reaches after a $d + d'$ delay is independent whether or not the state after d units of time has been recorded (time additivity),
- whenever a process may let d units of time to pass, there is an intermediate process for each $d' \leq d$ (time interpolation), and
- any process that can evolve for all durations $< d$ can also evolve for d (time closure).

These evolution assumptions have direct implications on the possible executions of timed CSP processes; they still allow unrealistic executions, such as timestops or processes that end with an infinite sequence of events. This explains the focus on well-timed processes, e.g. processes that have a determined delay between successive calls.

In **chapters 11-13** the focus is on the analysis of timed CSP: semantics (**chapter 11**), specification and verification (**chapter 12**), and refinement (**chapter 13**). Special emphasis is placed on the timed failures semantics for CSP; for P CSP process, the timed failures of P are denoted by $\mathcal{TF}[P]$. As we expect, a timed specification is a predicate $S(s, \aleph)$ on timed failures; a process P meets a specification $S(s, \aleph)$ (P *sat* $S(s, \aleph)$) if S holds of every timed failure of P:

$$\forall (s, \aleph) \in \mathcal{TF}[P]. \quad S(s, \aleph).$$

Since there are many similar specification patterns for safety, liveness, and common assumptions on the environment, several macros are constructed and used for exemplifying verification. To make the point clear, a mutual exclusion algorithm that uses delays is discussed at the end of chapter 12.

Chapter 13 plays the important role of making the picture round by explaining the more subtle connections between timed and untimed semantic models for CSP. The main idea is that timed descriptions contain more information than untimed ones, and thus can be considered *time-wise refinements*[7]. We can relate untimed and timed descriptions of processes, translate timed

[7]these are not refinements in the classical sense

specifications into untimed ones, prove refinements in the more simple untimed CSP and map back the results in the timed framework. The alternative method to add time to the untimed CSP by introducing an event *tock* to allow the passing of one unit of time is investigated in **appendix A**.

Appendix B gives an overview of the FDR (Failures Divergences Refinement) model-checker, software tool for automatic analysis of untimed CSP. FDR checks if one CSP process refines another, and analyzes whether a process may deadlock, diverge, or is deterministic.

Opinion

I find the book an excellent material for the reader interested in formal methods and concurrent systems. Rarely you can find a self-consistent book that builds a complex theory and still abounds in application-oriented examples; every concept is justified by examples ranging from simple models of real-life processes to detailed algorithms from concurrency theory. I find the examples even more useful since they gradually build in complexity, mapping the added functionality of the language.

If you are looking for a detailed exposition on the CSP language, then this is a book you must read. The book exhaustively covers the formal and less formal aspects of modeling concurrent and real-time systems using CSP; the presentation is up to date, benefiting from years of experience since the seminal work of Hoare ([Hoa85]).

However, it is sometimes difficult to focus on the essential points. I see the book as ideal for a reader with previous knowledge on the subject; the reader interested in the big picture would probably benefit more from reading [Hoa85], with the warning that it does not cover timed CSP.

References

[Fok00] W. J. Fokkink. *Introduction to Process Algebra.* Springer, 2000.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

Review[8] of Introduction to Languages, Machines and Logic: Computable Languages, Abstract Machines and Formal Logic.

Author of book: Alan P. Parkes, Lancastar University
Springer-Verlag 2002, xi + 351 pages, $67.75, Paperback

Reviewed by Robert McNaughton (mcnaught@cs.rpi.edu)
Rensselaer Polytechnic Institute

Introduction and summary: Most computer science undergraduate programs have a first course in theory, for which this book could serve as a text. Whether or not it is chosen depends on many things; by now there is an abundance of such texts, offering a wide range from which to choose. Reviews, such as this one, should be an aid to prospective instructors.

This book is divided into three main parts:

Part I: Languages and Machines; 6 chapters; 152 pages. Regular grammars, finite-state recognizers, context-free grammars, push-down recognisers, ambiguity, determinism and closure properties.

Part II: Machines and computation; 5 chapters; 113 pages. Finite-state transducers, Turing machines (including the universal TM, nondeterministic TM's and multitape TM's), Turing's thesis (usually called "Church's thesis" in the U.S.A.), the halting problem, and, more generally, the notions of computability and solvability.

Part III: Computation and logic; 3 chapters; 52 pages. Boolean logic, propositional logic, truth tables, quantifiers and predicate logic (including semantics) and computation with logical formulas. I should think that at many places where this book will be used, Part III will be omitted with logic perhaps covered in another course.

Commentary: Things are very clearly worked out in this book. An extravagant number of examples accompany the theoretical development, all presented in great detail, often accompanied by nice illustrative diagrams. For example, the subset algorithm (converting a nondeterministic finite-state recogniser to a deterministic one) is illustrated by its application to a certain four-state device; there are seven figures, each clearly and pleasingly drawn (pp. 63–66 in Chapter 4).

The promotional note on the book's front cover says, "Easy to read, nonmathematical approach," which means, I suppose, that every abstract formulation is accompanied by sufficient concrete exemplification.

Each chapter has from five to ten exercises, almost all of which are "cook-book exercises" in which the student must simply apply a well described solution recipe a few pages back. I have two criticisms here. First, I think the book should have about twice as many exercises so that, for each solution recipe, the instructor has more than one exercise to direct the student towards. Second, students in a course in theoretical computer science — even in the most elementary of such courses — should be given some assignments in which they most scratch their heads a bit before they can begin to work out a solution. Such experience is an important part of what a computer-science student needs to get out of a course in theory.

[8] ©2004 Robert McNaughton

I find this text to be acceptable for a theory course in an undergraduate computer-science curriculum. But the instructor who plans such a course must spend some time preparing suitable additional exercises.

Review[9] of
Algorithm Design: Foundations, Analysis and Internet Examples
by Michael T. Goodrich and Roberto Tamassia
Publisher: John Wiley & Sons, Inc.
720 pages, $74.95 new, $40.00 used, on Amazon
Paperback, 2001
Reviewer: Pavol Navrat, Slovak University of Technology in Bratislava, Slovakia

1 Overview

Design and analysis of computer algorithms and data structures has been a standard theme in computer science and computer engineering curricula since the very beginnings. It is also one of the most important ones, and one that has been most frequently elaborated. The book is intended primarily as a textbook for an Algorithms course.

2 Summary of Contents

The book is divided into four parts, each consisting of several chapters. The parts are titled Fundamental Tools, Graph Algorithms, Internet Algorithmics and Additional Topics. The authors do not provide a motivation or rationale for such a division, and vice versa, do not explain or justify inclusion of the chapters into a particular part, or indeed into the book itself. For most of the chapters this may seem superfluous or self- evident, also bearing in mind that themes for a standard course on algorithms (and data structures) have been discussed many times before. However, some of the choices, as we shall see, the authors made are worth discussing. The first part consists of five chapters. It starts with algorithm analysis, introducing asymptotic notation and methods of both asymptotic and experimental analysis of algorithms. Second chapter deals with basic data structures. Let us take a little closer look at how the topic is presented, since this may give us an idea about the pedagogic approach taken by the authors of this textbook. It starts with stacks and queues, introducing them informally as abstract data types. The notion of abstract data type itself is not defined, nor is explained a difference (if there is any intended) between the notions of an abstract data type and a data structure. The notion of abstract data type is used as a synonym for a collection of methods specified informally, for example:

push(o): Insert object o at the top of the stack.

It is not clear why an axiomatic specification was not considered. Immediately after such a specification, a simple array- based implementation follows. What remains unexplained, however, is the very concept of implementation of an abstract data type, the need of having some

[9]©2004 Pavol Navrat

implementing data type, and the necessity of choosing it. Of course, having once chosen the array as an implementing data type, the authors cannot skip over the assumption of a fixed upper bound on the ultimate size of the stack. They do elaborate that this implementation may be either wasteful or "crashing", but they choose not to go deep enough in the discussion to explain that it actually is not a correct implementation of the stack at all, since the operation push is not specified with an upper bound limiting it but is implemented so. For the time being, they make a forward reference to other ways of implementing stacks. Queues are treated similarly, since there are no other candidate implementing data types available at this point. Quite naturally, vectors, lists, and sequences are presented now, followed by trees, priority queues, heaps, dictionaries and hash tables. Third chapter is devoted to search trees and skip lists. Again, just to give an idea, it starts with ordered dictionaries and binary search trees. Binary search algorithm over an ordered vector is presented as well as tree search algorithm. The authors use a specific algorithmic language for describing the algorithms in the book. It is a pseudo-code, which is more compact than a corresponding fragment in a typical programming language would be. It is also easier to read and understand. Another regular feature of their approach is that the algorithm's presentation is followed by their analysis. The analysis is fairly detailed and formal, but explained in a very understandable way. Fourth chapter is devoted to sorting, sets, and selection. The reader finds here merge-sort, quick-sort, bucket-sort and radix sort. Fifth chapter presents some fundamental techniques: the greedy method, divide and conquer, and dynamic programming. That completes the first part. Second part is devoted to graph algorithms. In chapter 6, the graph abstract data type is introduced, and followed by data structures for graphs, algorithms for graph traversal, and directed graphs. Chapter 7 is devoted to weighted graphs, including the usual algorithms for single-source shortest paths, all-pairs shortest paths, and minimum spanning trees. The last chapter in this part is titled Network Flow and Matching. It presents an important problem involving weighted graphs called the maximum flow problem. Its inclusion is motivated by an example of a computer network. Third part is called Internet algorithmics. What is it? Let us look at the contents. It includes strings and pattern matching algorithms in chapter 9. More specifically, the Boyer-Moore and the Knuth-Morris-Pratt algorithms are studied, both published first in 1977. Then, tries are introduced. The Huffman coding algorithm used in text compression is presented. Chapter 10 is devoted to number theory and cryptography, or more accurately, to some schemes and algorithms for cryptographic computations, secure electronic transaction protocol and the fast Fourier transform. Chapter 11 deals with network algorithms: fundamental distributed algorithms, algorithms for broadcast, unicast and multicast routing. The book contains one more part devoted to additional topics. The authors have chosen three such ones: computational geometry, NP-completeness, and algorithmic frameworks. There are three different frameworks introduced: external memory algorithms, parallel algorithms, and online algorithms.

3 Opinion

The book is intended as a textbook for a standard course on algorithms (and data structures). There have been published many textbooks for this course. Anyone entering this arena is well aware of the difficulties arising from this fact. It is indeed very difficult to come with some novel approach. But only a strong argument along these lines can justify the project and keep it in a safe distance from becoming yet-another-data-structures- textbook. The authors have found a very good balance between formal treatment and readability. The choice of a pseudo-code as a presentation language for algorithms can be considered a positive (although definitely not a novel) feature. A consequent coupling of algorithm presentation with its analysis is also a very good thing. However, a better thing would be if an algorithm was not only analysed afterwards, but if complexity considerations were used beforehand to motivate the design of it (remember, the book's title is Algorithm design). In seeking the right amount of formality, the authors decided to work with abstract data types in an informal way. To some extent, this is a matter of taste. To some extent, this is a matter of consideration what can be gained by using e.g., the apparatus of axioms. For example, if analysis of algorithms was concerned also with their correctness, formal specifications would probably play a bigger role. Among other positive features, extensive exercises in each chapter should be mentioned. They are of three kinds: reinforcement, creativity, and projects. Moreover, there are included several Java implementation examples in the book. The scope of topics included is another positive feature (again, it is definitely a novel one). Text processing algorithms, network algorithms, some number theory and cryptography algorithms have been around in this kind of books for some time. To call them all Internet Algorithmics is an attempt, which may need some additional, perhaps deeper justification. On the other hand, some standard topics are treated in a less comprehensive way than one might expect: for example, the variety of sorting algorithms is quite narrow. In conclusion, the book provides a very well elaborated source to study algorithm design from.

Review[10] of
Theory of Semi-Feasible Algorithms
Authors of Book: Lane Hemaspaandra and Leen Torenvliet
Publisher: Springer, 148 pages, \$54.95, Hardcover
Author of Review: Lance Fortnow

1 Overview

A set A is feasible if we have an efficient algorithm testing membership in A. What if we have something close, say an algorithm that given two strings will tell us which is more "likely" to be in A.

P-selective sets capture this notion first defined by Alan Selman based on the notion of semi-recursive sets from computability theory. Formally A is P-selective if there is some polynomial-time computable function f such that for all x and y, $f(x, y) \in \{x, y\}$ and if x or y is in A then $f(x, y)$ is in A.

Not all P-selective sets are feasible or even computable. Let u be a real number between 0 and 1 and let B_u be the set of strings x such that $0.x < u$. The set B_u is P-selective by $f(x, y) = x$ if $0.x < 0.y$ and y otherwise. For different reals B_u and B_v we have $B_u \neq B_v$. Since

[10]©2004 Lance Fortnow

there are an uncountable number of reals and only a countable number of computable languages, there are non-computable P-selective sets.

P-selective sets are well-studied in structural complexity theory. Every P-selective set is in P/poly, i.e., computable in non-uniform polynomial time. In a weak converse every P/poly set is Turing-reducible to a P-selective set, though there are P/poly sets that are not themselves P-selective. If SAT, or any other NP-complete problem, is P-selective then P = NP.

This book focuses mainly on the complexity of P-selective sets and variations where the selector can be nondeterministic.

2 Summary of Contents

The first chapter introduces P-selective sets and various versions of NP-selectivity. They discuss the important results for these sets giving some proofs of the basic results and leaving some other proofs for later in the book.

The second chapter gives the connections between semi-feasible sets and advice. Advice is extra information provided to the machine that depends only on the input length. For example a language A is in P/poly if there exists strings a_0, a_1, $\ldots$ with $|a_n|$ bounded by a polynomial in n and a language B in P such that x is in A if and only if $(x, a_{|x|})$ is in B. This chapter presents results how, in many cases, the weakness of allowing selectivity can be replaced by advice. The book gives many refinements, limiting the kinds of reductions and amount of advice used. The chapter ends with the following pretty theorem: If SAT is NPSV-selective then PH $= \Sigma_2$. Informally that means that bad things happen if there was an efficient algorithm to take a formula ϕ on n variables to a formula ψ on $m \geq n$ variables such that if ϕ is satisfiable then every satisfying assignment for ψ sets its first n variables the same and those variables give an assignment for ϕ.

The third chapter describes lowness, a concept stolen from computability theory that describes that set of oracles a class could have without increasing its power. For example Low(NP) = NP∩co-NP since $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{co-NP}} = \mathrm{NP}$. They generalize the notion of lowness to extended lowness to capture nonuniform sets. They show that several of the semi-feasible sets sit inside some extended low classes.

The fourth chapter asks whether semi-feasible sets can be hard for various complexity classes. The answer is almost always no unless some bad thing happens. What bad thing happens depends on the semi-feasible notion, the class it is hard for (usually NP) and the hardness reduction.

The fifth chapter is entitled simply "Closures." For example P-selective sets are closed under complementation but not intersection. Only for some weak reductions are P-selective sets closed downwards. This chapter also discusses the relationship between self-reducible sets and semi-feasible sets, and the structure of the classes of languages either reducible to or equivalent to P-selective sets under various notions of reductions.

The final chapter talks briefly about related notions such as the membership comparable sets, multiselectivity and others.

3 Opinion

This book does a decent job in covering a limited topic: P-selective sets and nondeterministic versions of those. Nearly all the known results of these classes are covered in depth. I would have liked to see more depth on broader notions like membership-comparable than say a result that there is a set 2-tt reducible to a P-selective set that is not Turing-equivalent to a P-selective set, but that's just my taste.

A considerable background in computational complexity is needed to read this book. A single chapter added to this short book could have made the book stand alone to give an alternate introduction to complexity.

The book is heavy on notation though the authors make up for it somewhat by an appendix defining the terms used. I found the diagrams a bit confusing and in at least one case inaccurate.

The level of detail of the proofs is uneven. Some proofs are given in far too low level such as the three page proof of Karp-Lipton. Other times nontrivial results are left to the reader.

This book does give some nice introduction to some topics in complexity like the low and extended low hierarchies, nonuniform computation and of course the P-selective sets. To teach a course from this text would require a highly-motivated instructor who can give the intuitive ideas leaving the details to the book. The book would also serve as a reasonable reference for those doing research in this area.