INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

CSP++: An Object-Oriented Application Framework for Software Synthesis from CSP Specifications

by

William Bennett Gardner
B.S.E.E., Massachusetts Institute of Technology, 1974
B.Ed., University of Toronto, 1975

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. M. Serra, Supervisor (Department of Computer Science)

Dr. N. Horspool, Departmental Member (Department of Computer Science)

Dr. D.M. Miller, Departmental Member (Department of Computer Science)

Dr. G. McLean, Outside Member (Department of Mechanical Engineering)

Dr. D. Sciuto, External Examiner
(Dip. di Elettronica e Informazione, Politecnico di Milano, Italy)

Supervisor: Dr. Micaela Serra

ABSTRACT

One of the useful formalisms for designing concurrent systems is the process algebra called CSP, or Communicating Sequential Processes. CSP statements can be used to model a system's control and data flow in an intuitive way, constituting a kind of hierarchical behavioral specification. Furthermore, when coupled with simulation and model-checking tools, these statements can be executed and debugged until the desired behavior has been accurately captured. Certain properties (such as absence of deadlocks) can be proved, to help verify the correctness of the design.

To make the verified specifications executable in a practical sense, refinement to a programming language is required. In this work, an new object-oriented application framework is described which realizes the basic elements of CSP—processes, synchronizing events, and communication channels—in natural terms as C++ objects. In addition, a new software tool is provided to customize the framework by translating CSP statements into invocations of the framework elements. CSP specifications, thus reexpressed in C++ and compiled, form the control portion of a system, able to be linked with other software written in C++ that completes the functionality.

Examiners:

Dr. M. Serra, Supervisor (Department of Computer Science)

Dr. N. Horspool, Departmental Member (Department of Computer Science)

Dr. D.M. Miller, Departmental Member (Department of Computer Science)

Dr. G. McLean, Outside Member (Department of Mechanical Engineering)

Dr. D. Sciuto, External Examiner
(Dip. di Elettronica e Informazione, Politecnico di Milano, Italy)

Table of Contents

List of Tables

List of Figures

Acknowledgments

Warm thanks are overdue to many people, without whom this research would not have been accomplished:

First and foremost to my supervisor, Dr. Micaela Serra, for taking a chance on this "nontraditional-age student" (as the euphemism goes), for striving to model through her excellence in teaching, research, and care for students, what the academic life is supposed to be about, and for disdaining to be laid aside from her chosen course by mere life-threatening health challenges. She will always be an inspiration.

To Dr. Mantis Cheng, whose "Formalisms in System Design" course triggered the thinking that resulted in this work.

To the alphabet soup of agencies that financially or materially supported my research:

- NSERC: Natural Sciences & Engineering Research Council of Canada
- CMC: Canadian Microelectronics Corporation
- BC ASI: British Columbia Advanced Systems Institute
- SCBC: Science Council of British Columbia

and to Dr. Eric Manning, who encouraged me to apply to the Science Council for funding.

To ObjecTime Limited of Kanata, Ontario, for their generous donation of ObjecTime Developer software.

And finally, to some unsung heroes, system administrators Will Kastelic and Gary Duncan, who daily, unobtrusively, help make possible the department's computer work. Will deserves a special salute for cheerfully suffering thousands of pesky questions on C and Unix when I was a "green" grad student, struggling to get up to speed after seven years out of touch with the field.

Dedication

"Where is the LORD, the God of Elijah?"
2 Kings 2:14

This dissertation is dedicated to the memory of **Dr. Robert N. Thompson** (1914–1997), a distinguished Canadian educator, statesman, and politician. With pioneer farming roots in Alberta, he initially studied teaching and law, and would later take his doctorate in political science. During World War II, he served with the Canadian Army and Air Force, participating in the liberation of Ethiopia from Mussolini.

At the close of the war, Thompson responded to the call of Ethiopia's emperor to help rebuild that country, by moving his young family to Africa. He was just 29, and could not know that their lives would be intertwined with Africa for fifteen years. By the time they returned to Canada, Thompson had served Ethiopia in diverse roles, from Commander of Air Force Training to Associate Deputy Minister in charge of education. Known as "Mr. Education," mobbed by children and parents wherever he went, in one year he opened 26 primary and secondary schools. Shifting from government work to the Sudan Interior Mission in 1952, he headed SIM's educational program and directed a major leprosarium, writing of the latter in 1990 as "the most satisfying and fulfilling work I have done in my entire life." During this period, Thompson served as Advisor to the Ethiopian Government on Foreign Affairs, with assignments in Africa, the Middle East, and Europe.

It was serious illness that brought the Thompsons home to Alberta, with the six foster children they had adopted in Africa. He rapidly became a national figure on the Canadian political scene, as Member of Parliament from Red Deer, and federal leader of the Social Credit Party from 1961 to 1967. When Thompson retired from politics in 1972 and moved to British Columbia, he was a prominent Progressive Conservative, a ten-year member of the Parliamentary Committee on External Affairs, and entrusted with special missions to Congo, Southeast Asia, and Nigeria. During this period, Thompson taught political sci-

ence at what is now Wilfred Laurier University in Ontario, and had become involved with the then Trinity Western College in BC, as a member and then chair of the college's Board of Governors.

The last period of Thompson's life was dedicated to the furtherance of Christian education in Canada, by teaching political science at Trinity Western University, and working as Vice President of Development and Assistant to the President. Meanwhile, he was serving on the boards of several charitable organizations, including World Vision Canada, the Freedom Council of Canada, and Samaritan's Purse. He was often a consultant on business and education management and international development. By age 76, when he received the Order of Canada, one of this country's highest honours, his other awards and decorations, including the Star of Ethiopia, were too numerous to mention.

* * *

Now, it so happens that while I was on the campus of Trinity Western University this summer, writing up, I acquired Dr. Thompson's academic gown! The university archivist was simply relieving the collection of superfluous bulky artifacts, but to me, an immigrant, this gown will be a very special and intimate link with a bright name in Canada's own history, and a reminder that robust faith and dedicated service ought always to go hand in hand.

With God's grace, this gown will again join in convocation processions at the school that its former owner loved so much, and to which he devoted the final active years of his rich life on earth.

WBG
Langley, British Columbia
August 1999

CHAPTER 1

Introduction

Concurrent systems are well-known as a fertile source of design challenges [Rosc98]. The work in this dissertation concerns the development of new techniques and associated automated tools to support design and implementation in the realm of concurrent systems by way of formal methods and software synthesis. In this introductory chapter, the research will first be motivated, followed by an overview of the results.

1.1 Problems of concurrent system design

Concurrent systems often exhibit a high degree of complex interactions, both with their environment, as in the case of reactive real-time systems, and internally in terms of synchronization and communication among their constituent processes. One serious consequence is that designers have trouble guaranteeing system *properties*, whether this means the presence of good properties such as liveness, or the absence of bad properties such as deadlock.

The typical practices of traditional, as well as object-oriented (OO), software engineering, regardless which notations are employed for analysis and design, rely heavily on methodical testing to provide some assurances concerning system properties. However, for complex concurrent systems, it is difficult to rule out the possibility that some untested sequence of stimuli occurring in the field will expose, for example, a lurking deadlock situation. This provides a strong motivation to consider designing concurrent

systems using *formal methods.*

In contrast with informal design notations, design formalisms have strict semantics, whether, following Alagar's and Periyasamy's classification [Alag98], they are based on algebra, logic, set theory, relations, or some combination of these. Their underlying mathematical basis means that it is possible to ask questions about a formally-specified system, and answer the questions by carrying out a mathematical procedure. Because these procedures may be both onerous and error-prone when carried out manually, researchers have developed automated tools to facilitate the checking of formally-modeled designs. Their use enables software engineers to provide rigorous assurance about the properties of such systems that go beyond a warm feeling that "adequate testing" has been performed.

Leaving aside for the moment (until Chapter 2) the question of industry acceptance of formal methods, another important question arises: While it may be worthwhile to prove that a formally-specified design has the desired properties, who can say whether the properties carry over into the *implementation* created from the model-checked design? Since the specification notation cannot "run" on a target platform, since the notation is not itself a full-featured programming language, and since it is written at a relatively high level of abstraction, transformation (also called "refinement" [Hinc95]) into a detailed conventional program is required.

We know from experience that in the usual course of transforming a specification into an implementation, each step of manual refinement presents a fresh opportunity to introduce undocumented design decisions, and to cause the end product to diverge from its specified behavior. If we started with a formal model, the verified properties may well

become lost in the transformation. Another unhappy, but common, result is that the specification may become an isolated early design artifact. The more transformation steps are required, the less likely the specifications will ever be updated to reflect "as built" status, and the less value they will have to future maintainers.

If, on the other hand, the specification can somehow *become* the system, many pitfalls can potentially be avoided, including the abandonment of formal properties. We call a specification *executable* when there are tools to simulate it, reason about it, and, ideally, synthesize a realization using a chosen technology. It is fair to say that executable specifications are something of a Holy Grail for system designers.

We propose a two-part conceptual solution for concurrent system design that facilitates the use of a formal design notation, while at the same time avoiding the traditional problems of hand-transformation. This solution is based on the foundation of executable specifications, and it will now be described.

1.2 Conceptual solution via executable specifications

In the first subsection below, the strategy of using executable specifications is supplemented with another important element, that of *extensible* specifications. Together these elements make up a conceptual solution. This is followed by a statement of the research goals and the application domains to which we expect the results can be applied.

1.2.1 Two-pronged approach

The first thrust is to replace manual transformation with automatic translation from formal specifications to executable code. In doing so we can preserve formal properties. We

can also keep the specification in sync with the implementation by modifying the former and regenerating the latter, which is a sounder practice than what is usually done: modifying the latter and (possibly never) updating the former.

But automatic translation by itself is problematic because of some characteristics of formal notations, alluded to above:

- The specification, being at a relatively high level of abstraction, lacks the details needed for a full implementation.
- Moreover, the specification notation likely lacks even the semantical notions or syntactical constructs to denote those details.

These observations are approximately the same as saying that formal notations are not full-featured programming languages.

From here we can take either of two routes to close the abstraction-level gap in the pursuit of software synthesis:

1. We can extend the formalism's native notation by mixing in programming language-like constructs. This results in a hybrid notation or even a unique new language.
2. We can allow the formalism to play its major role in expressing high-level abstractions—such as hierarchical decomposition, control flow, synchronization, and communication—and provide, in addition, a "trap door" for stepping out of the formalism into a notation where the detailed, low-level operations can be expressed in a more conducive manner. This, in essence, means putting "hooks" into the formalism to accommodate extensions.

The big drawback to the first route is that tampering with the formal notation may, at best, make it incompatible with the model-checking tools we want to employ, and at worst, may "break" the formalism by introducing constructs that lack a consistent mathe-

matical basis. Thus, the second route is the one we will take. It neither tampers with the formal notation nor breaks the mathematical basis, provided that the "hooks" are suitably circumscribed in their effects. This constitutes the second thrust of our two-prong solution, that of making specifications extensible as well as executable. It is a necessary ingredient in our synthesis solution for concurrent systems.

We can now set forth our goals for constructing the solution outlined above.

1.2.2 Research goals

First, we want to start with a formal specification notation that is both *checkable* and *synthesizeable*. Checkable, here, means that there exists a well-developed suite of software tools that allow a designer to reason about a specification and verify its properties. Without this kind of model-checking support, the benefits of using a formalism are much more difficult to obtain, and our whole strategy loses its appeal. Synthesizeable means that the formal notation lends itself to conversion into an executable program, i.e., that it is a kind of executable specification. Formalisms that are largely systems of constraints, for example, may not be synthesizeable. If a model-checking tool supports simulation, i.e., "running" a specification, this is a likely indicator that its input formalism is synthesizeable.

Second, we want to develop a technique for synthesizing software from the formalism that can be executed on a target platform (characterized by some combination of processor and operating system) as "production code," apart from resource-hungry simulation tools. Achieving this goal is the main thrust of this work and takes up the bulk of this dissertation.

Finally, having made the observation above that for practical programming use a

formalism needs to be extended in some fashion, we want to provide a means of hooking procedural extensions into a formal control specification. The extension language should be a popular programming language, and must be compatible with the programs which are output by our synthesis tool.

1.2.3 Application domain

The goals just described, in principle lay out a general-purpose solution for concurrent systems design, not targeted to any particular domain of software applications. However, the choice of a particular formal specification methodology and the choice of a language for software synthesis and specification extensions—that is to say, the actual inputs and outputs of the automated tools—will bring some practical limitations that will be more or less suitable for diverse application domains.

Since the purpose of this research is not to produce a finished, marketable product, but rather to explore and demonstrate a proof-of-concept, it is acceptable to consider, within theoretical constraints implied by the goals above, choices based on prudential criteria such as technical familiarity, development cost, and personal interest. The selection of CSP, Communicating Sequential Processes [Hoar85], for input, and C++ for output is explained and justified in Chapter 2.

Given those choices, the application domain for this research in its initial form will be systems that are natural to model using CSP, including those with inherent parallelism, and targeted on hardware/OS platforms that support C++. As to the former, the primary commercial user of CSP so far seems to be the telecommunications industry, where, according to Formal Systems of Oxford [FSE], it has been effective in modeling commu-

nications protocols. Other application areas cited by Formal Systems include VLSI design, networking and data distribution, control, signaling, fault-tolerant systems, and human-computer interface. As for C++, its compilers and run-time libraries are ubiquitous.

Taking CSP and C++ together, the main targets that would probably be ruled out by this combination are hard real-time systems (because CSP, in its original form, lacks any notion of timing) and highly resource-constrained systems. The latter includes the subset of embedded systems with strictly-limited CPU ability, idiosyncratic processors for which no C++ compiler exists or compiled code is too uncompetitive with hand-coded assembly language, and/or small memory that cannot afford much heap space, or stack space for multiple threads. More will be said about our limitations in Chapter 2. It should be noted that some of these limitations can be reduced or eliminated by carrying out the prospective Future Work (see Section 7.2).

Having ruled out that group of candidates, we are still left with a wide range of computing platforms, from large-scale general-purpose systems down to embedded systems that are not too constrained. With the availability of our technique, it is possible that using CSP will become more popular with designers of systems of all sizes for which verification is a priority—for example, safety-critical systems—who had previously turned away from formal methods due to a lack of assistance with software synthesis.

1.3 Overview of results

This section provides a road map to the rest of the dissertation, as well as a broad summary of the results obtained.

In Chapter 2, we present the background on and rationale for our choice of the formalism CSP, and related work in the area of software synthesis.

The main novel approach that results from this research is the use of object-oriented application framework (OOAF) technology as a target for software synthesis. Background on OOAFs is also included in Chapter 2. In short, we automatically translate a CSP specification into source code for customizing the framework we call **CSP++**. A customized framework instance is then converted to an executable program by a conventional C++ compiler, and linked with user-coded extensions known as external routines, also written in C++. This design flow is depicted in Figure 1, with the heavy borders denoting the software components created by this research. When the customized framework code is run, the effect is of executing the original CSP specification, coupled with the C++ user extensions.

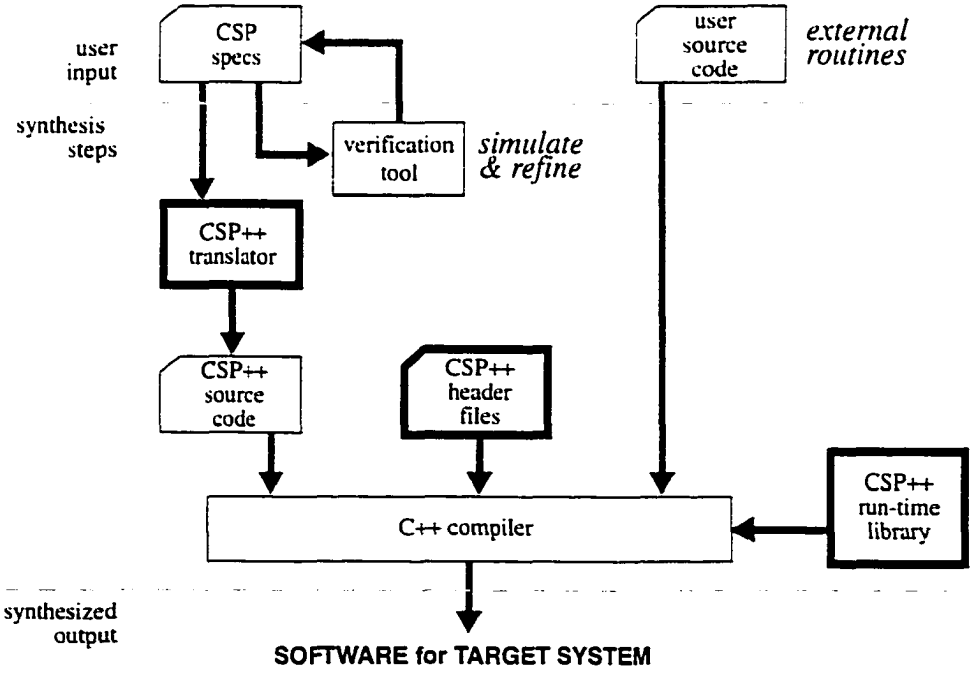


Figure 1: Use of CSP++ framework

In practice, we envision the CSP specification as forming a "control layer" in the layered system model illustrated in Figure 2. One can think of the CSP control portion as the "brain," with the user code forming the system's "limbs and organs." This figure also shows the capability of the user-coded external routines to provide an interface layer to packaged software modules—supplied perhaps in the form of other C++ class libraries—such as a database subsystem or OS facilities.

The key purpose of this work was to create a means of synthesizing software from CSP specifications. These results are presented in Chapter 3, which describes the OO architecture of the CSP++ framework, and in Chapter 4, which exhaustively lists the translations for the various CSP constructs into analogous C++ code based on the framework's components. These translations are carried out automatically by **cspt**, the tool that was created to customize a CSP++ framework instance according to a given CSP specification. This translator, fully documented in Appendix D, is vital for making any practical use of the software synthesis design flow.

Chapter 5 further illumines the CSP++ framework infrastructure by detailing its run-time operation as it implements the basic semantical features of CSP. Our solution is

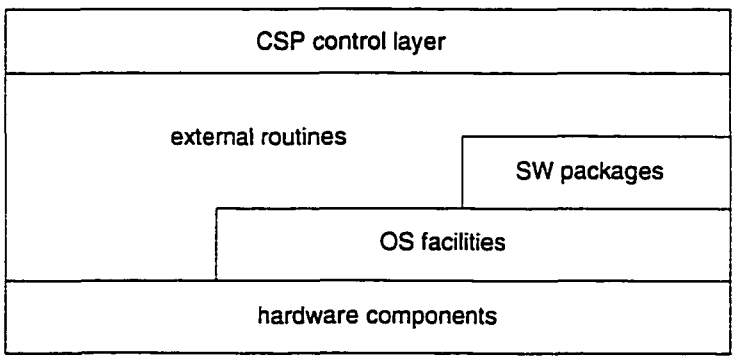


Figure 2: Layered system model

given for the critical problem of implementing multiprocess synchronization in the presence of the CSP deterministic-choice operator.

The CSP++ framework and cspt translator have been implemented, and currently run on two platforms. Chapter 6 gives the results of timing and memory measurements that were made on several test cases, and a comparison with a commercial synthesis tool based on StateCharts. Discussion of research results, conclusions, and possibilities for future work form the final chapter.

A case study based on a simplified disk server is introduced in Chapter 2 and picked up again in Chapter 6, where its translation and execution are explained. Source code for the case study and a user's manual for the tools are supplied in appendices.

In summary, this dissertation presents pioneering work in the areas of CSP synthesis and OO application frameworks. It is hopeful that with some additional development and optimization, this work could form the nucleus of commercial CASE tools and help popularize CSP as a design methodology.

CHAPTER 2

Background and Rationale

The purpose of this chapter is to introduce and explain the two key design choices in this research: first, the selection of CSP as the input formalism, and second, the use of a synthesis approach based on OO application frameworks, and the associated use of C++ as the implementation medium. In the course of this explanation we refer to related work and discuss the advantages and disadvantages of these two choices.

2.1 CSP

The usefulness of formal methods for system design is often disputed. In modern general-purpose software engineering texts, treatment varies from lightly touching on the Z formalism under "Advanced Topics" [Pres97], to devoting a few chapters to an overview of so-called "algebraic" and "model-based" methods, with a fuller look at Z as representative of the latter category [Somm96]. Both these texts highlight the controversial nature of formal specifications, viz "Formal specification on trial," a section heading in [Somm96]. Textbooks aimed at teaching formal methods to computer science students, e.g., Alagar and Periyasamy's *Specification of Software Systems* [Alag98], are still fairly rare.

On the one hand, proponents point to the superiority of formal notation over natural language specifications for reducing ambiguity and the typical proliferation of alternate interpretations amongst software developers, test engineers, and other participants. Math-

ematical analysis of formal specifications can result in provably correct software behaviour, which is undeniably important for safety-critical applications such as nuclear power plant control systems, medical devices, and avionics. The prospect of significant savings in design, implementation, and validation stages is held out, at the expense of additional investment at the specification stage.

On the other hand, opponents object to what they maintain is a confusing use of abstruse mathematical notation, which practitioners are reluctant to master, and additional engineering process steps that commercial developers are reluctant to budget for. Sommerville helpfully points out that proponents tend to argue about claimed technical improvements, while opponents often respond on the basis of unjustified costs [Somm96]. A balanced perspective is given in the article "Formal Methods: Promises and Problems" [Luqi97].

It is not the purpose of this research to take sides in this debate. We take it for granted that some people will be enthusiastic over incorporating algebraic notation into a system's specifications, while others will demur. Rather, our interest is in providing a new tool that can make the adoption of one formal design notation, CSP, more practical and attractive. By building an avenue from model checking and simulation to software synthesis, we broaden the usefulness of CSP.

2.1.1 Background on CSP

The classic work on CSP, *Communicating Sequential Processes*, was written by its inventor Tony Hoare [Hoar85]. It methodically covers the fundamental principles of processes, concurrency, and communication, and introduces formal techniques by which models

expressed in CSP may be logically verified. We note that, over the years, CSP notation has not been standardized, and that new operators can be invented, thus *dialects* of CSP have evolved.

The abundance of algebraic and set notation in Hoare's original book may seem overwhelming. In that case the following new book by A.W. Roscoe will be a better choice: *The Theory and Practice of Concurrency* [Rosc98]. As well as being designed as an undergraduate textbook, with a gentler ramping up of the math, it has been fleshed out with more case studies. It also has the advantages of incorporating recent research, and of using a dialect of CSP compatible with the commercial FDR model-checking tool from Formal Systems (more about FDR below).

Recently, Michael Hinchey and Stephen Jarvis have contributed a book [Hinc95] that features updated notation conventions. It should be noted that this book has come under heavy fire from one CSP expert, Bryan Scattergood of Formal Systems, for having "far too many technical errors" [Scat95], and should therefore by used with caution. Nevertheless, the authors have been associated with the seminal Programming Research Group at Oxford University, whose CSP archive [CSP] is a good starting point for exploring CSP on the Internet.

One more worthwhile source is Gajski et al, *Specification and Design of Embedded Systems* [Gajs94], which surveys CSP in the context of many other alternative methodologies. It also gives an overview of StateCharts, which we use in the next section to introduce a case study in a graphical manner.

Simply put, each statement in a CSP specification is the description of a *process*.

The process engages in a sequence of named *events*, which may include point-to-point communication with another process via a nonbuffered, unidirectional *channel*. The set of all events that a process may ever engage in is called its *alphabet*. These may correspond to real-world occurrences such as sensor input, device actuation, and so on.

Things get interesting when processes define themselves in terms of other processes, including several processes running in parallel. Then, the formalism provides for interprocess synchronization each time an event occurs that is in their common alphabet. This also implies that processes synchronize around channel communication. CSP statements can thus be used to model a system's control and data flow in an intuitive way, constituting a kind of hierarchical behavioral specification.

An example will make this easier to follow. The simplified disk server is one to which we will return later.

2.1.2 Disk server case study

The CSP notation we use here is that which is accepted by an in-house verification tool, **csp12**. It is mostly identical with the notation found in [Hinc95]. Csp12 was written in Prolog by Dr. M.H.M. Cheng, Department of Computer Science, University of Victoria, BC. To rigorously follow csp12 input conventions, we should use "::=" in place of "=" and "->" in lieu of "→" in the sample statements, and terminate each statement with a period. The source code in Appendix A is true csp12 code.

First consider Figure 3 which uses StateCharts [Gajs94] to visually portray DSS, the Disk Server Subsystem, interacting with C(*cl*) standing for multiple clients. We write the CSP for a two-client system starting from the complete system view:

```
SYS = ( DSS || (C(1) ||| C(2)) )^{ds,ack(1),ack(2)}
```

This states that the system `SYS` is defined as the parallel composition of the disk server DSS and two client processes, `C(1)` and `C(2)`. Here, the parenthetical notation should be thought of as machine-readable subscripting: C_1 and C_2. Parallel composition

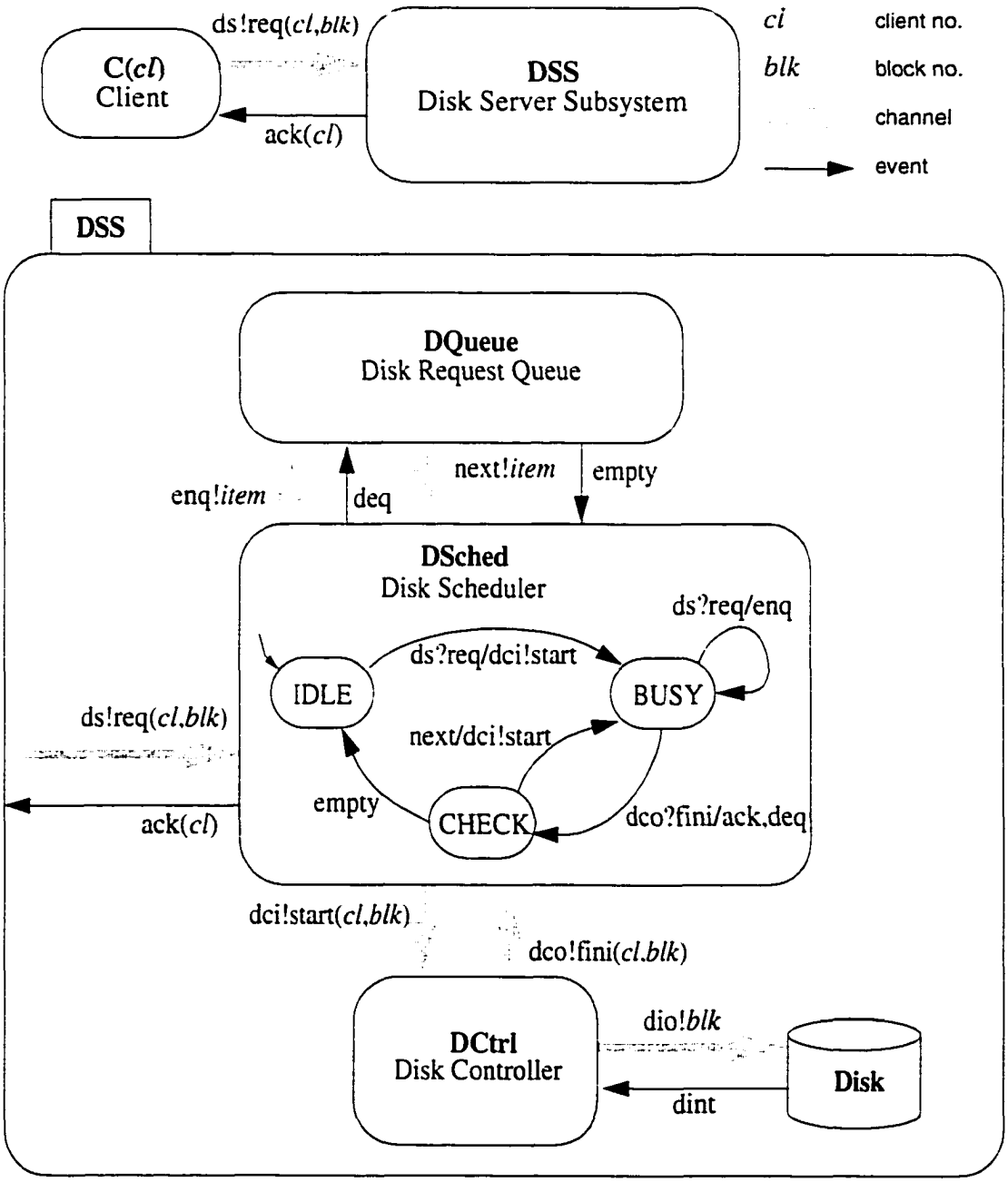


Figure 3: Disk Server StateCharts

of concurrent processes is expressed with the symbol "||" plus an outer caret "^" that explicitly denotes the set of events on which the processes will synchronize. The synchronization set includes channel `ds`, over which a client communicates a request, and `ack(`*cl*`)`, the acknowledge event to client *cl*. The clients are composed using the symbol "|||" which stands for interleaving; that is, they run concurrently but they do not synchronize with each other.

Strictly speaking, it should not be necessary to list the synchronization set, since CSP defines that any events in the common alphabet will implicitly cause synchronization. However, in practice it is difficult for simulators to derive the intersection of alphabets, and easy enough for the specifier to write it out. From the software engineering standpoint, an added benefit of this explicit notation is that these events change from being invisibly implied elements of the specification to being visibly documented.

The disk server is also defined as a number of subprocesses, corresponding to the four inner components of the DSS StateChart in Figure 3:

```
DSS = ( (DSched || DQueue)^{enq,deq,next,empty}
      || (DCtrl || Disk)^{dio,dint} )^{dci,dco}
```

The disk request scheduler `DSched` is composed with the queue, `DQueue`. Their synchronizing events concern the enqueuing and dequeuing of requests. The disk controller `DCtrl` is shown in this simplified model composed with a dummy process standing for the actual disk drive.

The scheduler `DSched` in Figure 3 is drawn as a state machine, and these next statements will show how CSP can accommodate this:

```
DSched = DS_idle
```

```
DS_idle = ds?req(_cl,_blk)→dci!start(_cl,_blk)→ DS_busy

DS_busy = dco?fini(_cl,_blk)→ack(_cl)→deq→DS_check
        | ds?req(_cl,_blk)→enq!req(_cl,_blk)→DS_busy

DS_check = empty→DS_idle
         | next?req(_cl,_blk)→dci!start(_cl,_blk)→DS_busy
```

The specification for `DS_idle` illustrates two constructs. First, `ds?req(_cl, _blk)` means that the process waits for input on the channel named `ds`. Input is denoted by the symbol "?" followed by a variable. Similarly, output is shown with "!" followed by a value. Here, channel `ds` receives the complex datum `req` which is made up of the client number and block number. These names `_cl` and `_blk` function as local variables for the process. The right arrow is a transition to the next event in the process, the output of the `start` datum on the channel `dci`. After this, the `DS_idle` process continues as the process `DS_busy`, in effect performing a state transition to IDLE.

`DS_busy` illustrates deterministic choice, which works like this example:

```
P = a→Q | b→R
```

`P` has a choice. If event `a` occurs, `P` will continue as process `Q`, but if `b` occurs, it will continue as `R`. (If neither occurs, `P` will not proceed at all). Looking back to `DS_busy`, we see that if the scheduler hears from the controller that the disk has finished a request (the input `dco?fini`), it will acknowledge the appropriate client and enter the CHECK state. Otherwise, if it gets a fresh request from a client (`ds?req`) in this BUSY state, it will enqueue the request and remain BUSY.

The CSP for the controller and disk are simple sequences of events:

```
DCtrl = dci?start(_cl,_blk)→dio!_blk→
              dint→dco!fini(_cl,_blk)→DCtrl
```

```
Disk = dio?_blk→dint→Disk
```

DCtrl waits for a start request on its dci channel. The start datum contains the client number and disk block. The controller sends the block request on its output channel dio and then waits for an interrupt event dint. On the line below, the Disk process simulates inputting the request and outputting dint. Thereupon, DCtrl signals completion by sending the fini datum on channel dco. The process continues as itself (DCtrl= ... →DCtrl), which specifies a loop (*not* recursion, as one might imagine).

The disk request queue (internal details not shown in Figure 3) is more interesting and shows the last of the CSP notation to be introduced here:

```
DQueue=((DQ(0)||BUFF)^{left,right,shift} )\{left,right,shift}
DQ(_i) = enq?_x→left!_x→shift→DQ(_i+1)
        | deq→( ( if _i=0 then empty→DQ(0) )
                    + fix X.( right?_y→next!_y→DQ(_i-1)
                                | shift→X )
              )
BUFF = CELL |> CELL |> CELL
CELL = left?_x→shift→right!_x→CELL
```

The queue process is described as a buffer (here only 3 cells) composed with a subscripted process DQ_i, where i denotes the number of items currently in the queue. Each CELL process receives a datum on its left channel and, after being told to "shift", delivers it on its right channel, and then continues being a CELL. The symbol "|>" is a special kind of parallel composition, which can be defined in terms of other CSP operators, used just for pipelines. P |> Q has the effect of making P's right channel synchronize with Q's left channel, so that data is passed from P to Q. The entire BUFF pipeline has left and right channels to communicate with DQ_i, and can be told by DQ_i

when to `shift`. The set of events prefixed by backslash "\" will be made local to this process `DQueue`, and not visible to any process with which it may have been composed at a higher level (this is called hiding or concealment).

As for DQ_i, when it gets input on `enq` it enqueues `_x` on `BUFF`'s `left` channel and does a `shift`. When it gets a `deq` event, then it faces a choice ("+" is the general choice operator): if there are no items in the queue (shown by a zero subscript), the `empty` event occurs and the process continues as DQ_0. Otherwise, a subprocess `X` is declared ("fix" is a way of putting a process in-line): If `BUFF`'s `right` channel yields up an item `_y`, it is passed out through channel `next` and the process continues as DQ_{i-1}. Otherwise, a `shift` is ordered and the subprocess is repeated.

We can make two observations about this buffer specification: (1) The shift action in `CELL` is actually superfluous and is given to illustrate more CSP++ constructs. (2) It is evident that CSP is hardly an optimal way of implementing a simple FIFO buffer. Here, the strength of CSP in specifying control flow gives way to its weakness in manipulating data. An improvement would be to implement the buffer with a user-coded external action.

There are a few other CSP constructs that we have not encountered in this example. These include event renaming (which is how pipelines are implemented) and nondeterministic choice, which is useful for keeping specifications at a high level of abstraction, though not for actual implementation. As noted above, it is permissible in CSP to invent new operators. For example, the "fix" operator is a convenience notation supported by the in-house tool **csp12**. Variants of CSP also exist, one of the most useful for real-time systems being Timed CSP [Davi92], which adds timing constraints to the arrows between

events.

Now that we have the specification, what's next? Since CSP specs are executable, we can turn to a simulation tool to run it. At the University of Victoria, csp12 will accept the above syntax. A more sophisticated "industrial strength" simulation tool, called FDR [Rosc94], is available commercially through Formal Systems of Oxford [FDR]. These tools can also perform *model-checking*, which is a major virtue of formal methods having precise algebraic rules. In general, three properties can be checked, requiring various amounts of run time depending on the complexity of the specification: deadlock, livelock (infinite loop), and equivalence. The last property means that if we have two specifications for a process or system (perhaps one, P', is intended to be a "better" version of P), we can prove whether they are indeed equivalent. See Chapter 3 of [Hinc95] for details.

The difference between simulation and model-checking is this: When a CSP system is simulated, one of many possible paths through the specification will be followed, and the path will be logged in the form of a *trace*, that is, a sequence of executed events. A number of successful simulation runs no doubt builds confidence in the correctness of the specification, especially for simple systems, but does not by itself guarantee that pitfalls are not lying down paths that have not been exercised. Model-checking, on the other hand, conducts an exhaustive analysis (which is why it tends to be expensive in computation time) of all possible traces, in order to verify that certain desirable states can be reached under specified conditions, and that no harmful states can occur. This is an advantage that formal methods can afford compared to conventional programming.

2.1.3 Why CSP?

In terms of an input formalism for this research, our stated goals (Section 1.2.2) require that the formalism be checkable and synthesizeable. A few algebraic specification languages meet these criteria, including CCS, Calculus of Communicating Systems [Miln95], and ACP, Algebra of Communicating Processes [Berg85]. The special appeal of CSP came from the availability of the free in-house tool (csp12, described above), and the factor of greater familiarity. The existence of the sophisticated commercial tool, FDR, meant that a path for applying this research in industry could potentially be followed up. Generally speaking, this same work could have been done with an alternate formalism; however, we are not aware that anyone has done so. Furthermore, it would be possible to adapt our framework for code generation—in particular, the translation front end—to utilize an alternate input language, as noted under Future Work (Section 7.2.5).

The main disadvantage that arises from using CSP is its lack of the notion of time. We have already indicated (in Section 1.2.3) that this limits the application domain of our technique. On the other hand, Roscoe argues that CSP's handshaken style of communication is a good means of abstracting away the timing element, and that protocols that do not rely on timing for correct behaviour can be more robust [Rosc98]. This is not by any means to deny that timing is a requirement in some systems, and it would indeed be possible to extend this work to implement a timed variant of CSP, as described under Future Work (Section 7.2.3). However, we have intentionally left that more complex issue for later.

2.2 Object-oriented application frameworks

The second key underpinning of this work is the decision to involve OO application frameworks. This is not just an unthinking reflection of the "OO craze," but actually represents a fresh approach to software synthesis. We employ an OOAF as a high-level synthesis target, as opposed to the customary approach targeting assembly language or high-level language source code. The choice was intended to pose the research question of whether this approach would be worthwhile, and to probe its strengths and weaknesses. As with the choice of CSP above, a rationale will be presented following a brief background section.

2.2.1 Background on OO application frameworks

Object-oriented application frameworks are a fairly new development in the world of OO software engineering, and not a lot has been written about them yet. Budd defines an OOAF as "a set of classes that cooperate closely with each other and together embody a reusable design for a general category of problems" [Budd97]. The first book on this emerging technology, *Object-Oriented Application Frameworks*, writes in similar terms of classes "with a built-in model of interaction," constituting "a programming environment for vertical applications" [Lewi95].

This version of the recurring OO theme of *code reuse* represents, in a way, an extension of class libraries, a venerable OO practice, and design patterns [Gamm95], the enthusiasm that just preceded frameworks. The contrast is instructive:

- Class libraries package up sets of utility functions in precoded OO format, for use in any kind of system that happens to need those functions.

Utilization is via instantiating the classes and/or subclassing (inheriting) them for refinement purposes.

- Design patterns, on the other hand, do not come precoded. They are generic solutions to common design problems, laid out in terms of cooperating classes, and are utilized by copying the models and filling in the details according to one's own application.

Like class libraries, frameworks are comprised of a set of precoded classes, but unlike class libraries—and like design patterns—the classes were all designed to cooperate to implement a particular kind of application. Frameworks are specific enough that they are not amenable for use in arbitrary applications, yet they are general enough that a degree of customization is possible. Another useful way to view a framework is as "a semicomplete application that contains certain fixed aspects common to all applications in the problem domain, along with certain variable aspects unique to each application generated from it" [Srin99]. These variable, or customizable, aspects have come to be known as *hot spots*.

Examples in [Lewi95] are mostly from the world of systems programming, and concern areas such as operating system I/O and graphical user interfaces, including the well-known Microsoft Foundation Classes. In contrast, when the *Communications of the ACM* special issue on frameworks was published two years later [Faya97], it featured frameworks drawn from diverse industries, from multimedia to semiconductor manufacturing. In another two years, growth of the technology has been sufficiently explosive that Wiley and Sons is issuing a three-volume set with these titles:

- *Building Application Frameworks: Object-Oriented Foundations of Framework Design*

- *Implementing Application Frameworks: Object-Oriented Frameworks at Work*
- *Domain-Specific Application Frameworks: Frameworks Experience by Industry*

To our knowledge, CSP++ is the first application of this technology to software synthesis, so it is interesting for that reason alone. CSP++ is featured in chapter 9, co-authored with Dr. M. Serra [Gard99b], in the book *Implementing Application Frameworks: Object-Oriented Frameworks at Work* [Faya99], part of the new Wiley OOAF set.

2.2.2 Why a C++ OOAF?

The problem of software synthesis requires generating code that will run on a target platform consisting of a designated CPU and OS combination. Unless the platform is extremely limited, there will normally be a range of possibilities as to the *level* of source code that can be generated for it. These levels are portrayed, albeit simplistically, in Figure 4. Generally speaking, the difficulty of code generation is directly related to the logical "distance" between the input abstraction (in our case, CSP) and the level of the code generation target. A greater distance results from a semantic mismatch between the abstraction's model of computation and that of the target language. For example, if the abstraction is a dataflow model, then assembly language for a typical von Neumann CPU represents a relatively large distance. If the semantic mismatch is not great, then syntactic differences would carry more weight.

Therefore, assuming the input abstraction is not very primitive, generating assembly code for a bare processor with no executive is by far the hardest job. Such a code generator must concern itself with low-level issues such as register allocation and memory lay-

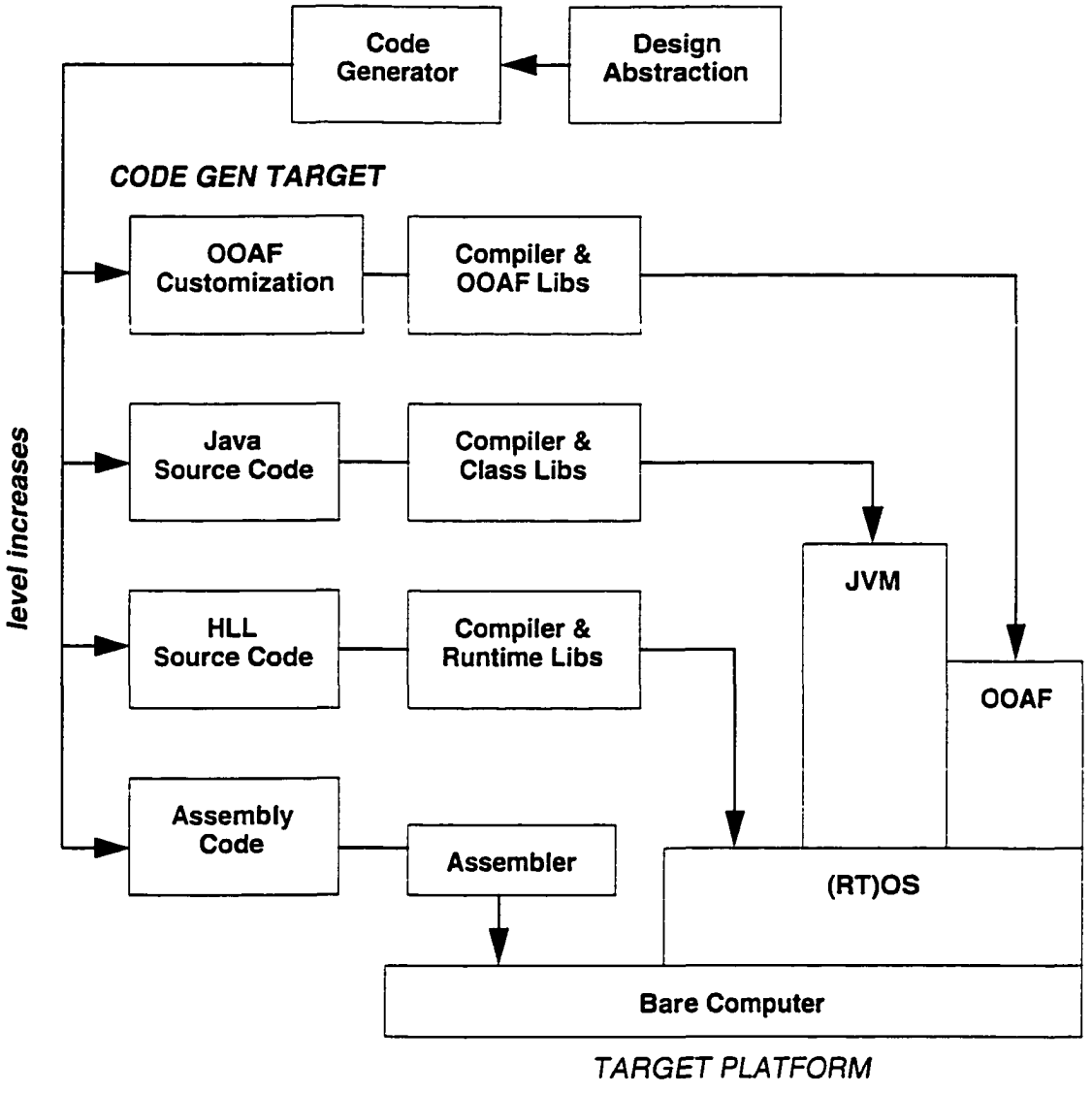


Figure 4: Levels of targets for code generation

out. In contrast, moving up one level (in Figure 4) and outputting high-level language (HLL) source code, say C, would be much easier, because it leverages the existing well-developed technology resident in the C compiler, and the services of the compiler's run-time library and underlying OS.

In recent years, Java has presented another option. It is portrayed in Figure 4 as a higher-level target because it relies on a virtual machine (VM) and extensive class librar-

ies.

We show "OOAF" as a still-higher level target. This is because the most primitive components that the code generator must be concerned with can be intentionally designed to be very close to those in the design abstraction itself. Thus, the translation distance is shortened, such as the distance between the German and Dutch languages, for example, as compared to, say, English and Chinese.

We also picture OOAF as smaller than the Java VM. This is meant to show that while the JVM has to be general-purpose, with a resource footprint to match, an OOAF contains only what is necessary for its specific mission, in this case, emulating communicating sequential processes.

To utilize an OOAF approach, we create a set of classes representing the basic components of the CSP paradigm, and make them cooperate to perform CSP's basic functions: process creation, event execution, choice, interprocess communication, synchronization, and so on. Indeed, it was pointed out by a reviewer of [Gard99b] that a CSP specification is itself already a kind of "framework," at least in the way we are using it, in that the abstract events are the "hot spots" which are customized when associated with user code.

Commercial frameworks are typically large and complex, and customization is to a large degree manual. In contrast, the CSP++ framework is small and relatively simple, and its customization—which occurs every time a CSP specification is translated—is largely automatic.

Using an OOAF approach naturally implies using an OO programming language.

We use C++ because it has compilers known to generate efficient code, and because features such as polymorphism, operator overloading, templates, and preprocessor macros can all be pressed into service in order to create a framework customization methodology, i.e., CSP++ statements, largely similar to CSP syntax. The effect is that the C++ compiler is enlisted to do the heavy work of assembly code generation, after the CSP++ translator has done the comparatively light work of producing compiler-ready C++.

Another benefit of packaging CSP++ as an OOAF is that the multitasking model is not too difficult to change, which may be necessary for porting to a different OS. Ideally, one should need only to alter the task base classes, and leave the rest of the framework intact. Our experience in porting to a new platform showed that this was largely the case (see Section 3.5 "Platforms" and Section 3.5.3 "Lessons from Linux port").

To be sure, using C++ brings drawbacks for some potential application areas. From the viewpoint of more resource-limited embedded systems, C++ seems to make lavish use of resources, particularly memory. Multithreaded C++ is even worse, because each thread requires its own stack, as well as the heap for dynamically allocated variables.

Nonetheless, we believe the benefits of an OOAF approach outweigh the drawbacks. One considerable benefit was that by erecting a high-level code generation target, and thereby shortening the translation distance, the entire project became tractable for one person over a reasonable time frame. Furthermore, the relative ease with which the essential features of CSP were implemented using this approach eclipses the meagre results obtained by some earlier work (CCSP) that attempted to translate CSP to a lower-level target (the C language). That and other related work will be presented in the next section.

2.3 Related work

In relation to our goal of software synthesis from CSP, there have been some efforts at making CSP specifications run as programs. Historically, the programming language occam has been derived from CSP, and Chapter 9 of [Hinc95] shows how to convert from CSP to occam, but they also acknowledge that it is a "very specialized language intended for implementation on transputers." Our goal is quite different: We wish to translate CSP into a popular language that will make it easy to combine with other code that fills out the functionality of the system.

Code generation has been done to some extent for the C language. The CCSP tool [Arro94] provides a limited facility for translating a subset of CSP into C, but it does not directly support the key parallel composition operator (||). Instead, each CSP process becomes a heavyweight UNIX process, and channels are implemented as UNIX sockets. In contrast, our approach supports the full functionality of concurrent composition, and is implemented using threads, thus making it practical for a larger range of applications.

For Java enthusiasts it is worth noting that the JavaPP (Java Plug & Play) Project has created a set of classes called CJT, Communicating Java Threads [Hild97], which are designed to bring CSP-style synchronization and communication to Java programs. Again, this represents a different goal from ours, but does open up an avenue for converting CSP to Java. We have declined to take this route, partly because of the considerable overhead entailed in running a Java Virtual Machine. More pragmatically, when this research commenced in 1995, Java was still at too early a stage to be seriously considered. Nonetheless, the Java option may be worth exploring under future work, especially in light of the recent development of native bytecode processors, e.g., Sun's PicoJava pro-

cessor core [McGh98].

There is another well-developed derivative of CSP and CCS called LOTOS [Logr92]. It is similar to occam in being a full-featured programming language. In addition to the process-algebraic aspect, LOTOS also incorporates a data-algebraic subset based on abstract data types, and it compiles to executable code. The language has been standardized (ISO 8807), and is in use, particularly in Europe, for design of distributed systems and protocols. In conjunction with using LOTOS as a specification language for hardware/software codesign [Carr96], synthesis tools for translation of LOTOS to C and VHDL have been created. As with occam, LOTOS represents a different direction than our work—that of utilizing an entirely new language, albeit based on a design formalism.

2.4 Objections and rejoinders

In light of the background and related work above, this section further discusses the rationale for our approach by means of raising and responding to objections.

1. *Since there are already compilable programming languages based on a formal model, why not just write software in one of those?*

That approach is valid, and has been taken in the cases of occam and LOTOS, where one or more formalisms was expanded into a full-featured programming language. Our approach starts with two assumptions: (a) at the high level there are going to be "specifications" in any case; and (b) at the implementation level, people prefer to code in familiar, popular languages.

Our suggestion is: First, learn to write those specs in CSP, at least for those parts of the target system for which CSP is a natural expression. Use model-checking tools to

evaluate them, and our tool to translate them into C++. Then use C++ to fill in the CSP events as user-coded procedures, which will form the bulk of the total code. Asking some upper level designers to write CSP is very different, organizationally, from asking every programmer to become proficient in occam or LOTOS. After all, CSP is not one of the very obscure or abstruse formalisms, so even the programmers who do not master it can understand specifications written in it.

2. *Why attempt to implement a concurrent system in C++, a language that lacks a built-in model of concurrency?*

The overhead of Java, which does possess built-in concurrency has been cited as a drawback above. We could, alternatively, have turned to a concurrent programming language, but this would have defeated the purpose of involving a common, popular language. In fact, it is not difficult to provide concurrency for C++ programs by utilizing a POSIX threads package [Lewi98], or other suitable class library, as we have done.

3. *What is the advantage of mixing two languages, CSP and C++, into a single system implementation?*

The point of using CSP at all is as a powerful *specification* tool, not a programming language per se. However, by applying our research, we can directly make it executable. Therefore we can maintain that portion of the system's code arising from the CSP by directly maintaining the spec, and regenerating the code whenever the spec is changed. Note that this is the opposite of the usual software engineering practice (i.e., change the code, and then hopefully update the spec).

Another advantage that falls out of this approach is that the design can be modularized in two places:

First, and most obviously, we can modularize in terms of abstract CSP events. That is, a programmer could be assigned to implement a particular event in C++. The CSP process context in which the event is invoked represents the spec, and if the programmer wants to change that, it can be handled as a spec change. This point is important, since the change may affect other modules or even the system behaviour. In that case, if the CSP specification is under configuration control, it should be modified by a higher-level designer, reverified, and resimulated.

Second, since CSP processes can be expressed in terms of other processes, not solely in terms of events, a process can become a "module" as well, with a specified interface of channels and events. Such a process can be initially coded as a "stub," that simply goes through its communication handshake and exits. The `Disk` process in the DSS example is just such a stub. Implementation could proceed by refining the process into subprocesses, until finally the level is reached of individual events having the desired degree of complexity.

4. *If there is a problem in the translated CSP, how will a programmer trace it back to a particular CSP source statement?*

This is the same problem that arises with a compiled language such as C++. There, the practice is to add numerous print statements, or else run the program in a debugger, setting breakpoints, etc.

We have provided two debugging features in CSP++: First, one may run the compiled system with the command line trace (-t) flag (see Appendix B.4). This will cause the run-time framework to log all the events as they are executed, annotated with the name of the process in control at that moment. Second, the translated CSP++ code is

itself standard C++ source code, not assembly language. The target system can easily be run in a thread-aware symbolic debugger such as **gdb**, breakpoints can be set, the code can be stepped through, variables inspected, and so on. Moving back and forth between the translated CSP statements and the user-coded C++ procedures is completely seamless. If the translator is invoked with the source (-s) option (Appendix B.2), the CSP input source code will be interleaved with the C++ output as comment lines, making it easy to associate them with the corresponding C++ statements during debugging.

5. *By adding user-coded procedures to the semantically-limited primitive communication events of CSP, doesn't this "break" the formalism? Does this render our analysis, simulation, and tool-based model-checking ineffective?*

To put the question another way, are we achieving a mere veneer of formalism? The answer hinges on what the user-coded procedures are allowed to do. As long as the procedures never communicate or synchronize with one another "behind CSP's back," so to speak, the encompassing formal model is maintained. As far as the model is concerned, its abstract named events are strictly atomic and of indefinite duration, and what an event's semantics are in the context of the computer system is irrelevant.

While it would probably be legitimate to allow any or all events in a CSP specification to be associated with user-coded procedures, in order to prevent confusion and make the implementation of synchronization straightforward, we imposed the following rules:

1. Events used for interprocess synchronization must be dedicated to that purpose, and are not allowed to have associated user code.
2. Consequently, interprocess communication must be performed strictly via CSP channels.

Note that these rules do not prevent user procedures from participating in choices, nor from implementing channel semantics by performing their own I/O operations, e.g., to communicate with the system's environment. That would be the case with, say, the `DCtrl` process in the DSS example, if the dummy `Disk` process is removed. Furthermore, user procedures can even safely communicate with one another (e.g., through static variables) as long as they are only ever invoked by the same process.

The above arrangement, embodied in CSP++, offers the "best of both worlds": a formal method for specifying high-level system behaviour, and a popular programming language, C++, for implementing detailed low-level behaviour. This is made possible by the ability to automatically translate the former portion into C++ to link with the latter.

CHAPTER 3

The CSP++ Framework

This chapter describes the architecture of the OOAF, which forms the run-time system of CSP++ and is customizable, via the translator, to execute any given CSP specification. It is the code generation target for the cspt translator and forms the heart of our novel software synthesis solution.

First, we present a lengthy discussion of architectural issues that arise in building a software system to emulate the semantics of CSP. This ranges over the choice of objects that will serve as code generation targets, and the dynamic aspects of CSP, including process scheduling, interprocess communication, and the binding of symbols in the process environment.

Next, the framework's design goals are presented, which influenced certain implementation choices. Following this, the detailed design description proceeds, in the usual way for OO designs, by walking through the class hierarchy. Then, the means of integrating user code with the framework is outlined. Finally, a description is given of the two platforms on which the framework has been implemented to date.

A detailed walk-through of run-time operation is deferred until Chapter 5. This will be more meaningful after the representation of CSP statements in terms of the OOAF has been presented in Chapter 4.

3.1 Architectural issues

These issues have to do with the challenge of replicating the CSP computation model in a C++ framework which is capable of being customized to act in a fashion specified by a set of CSP statements. The sections below, for the most part do not describe the details of our implementation, but instead highlight the theoretical issues that any form of implementation has to grapple with.

3.1.1 Process scheduling

The first critical issue concerns the run-time flow of execution in a customized framework instance. To start with, the model of computation inherent in CSP maps directly into multitasking or multithreading, where each CSP sequential process becomes an individual task or thread, having the appearance of sequential execution. There is no reason why we cannot use conventional multithreading as the basis for our implementation, and it will be convenient to do so. Just as the execution of CSP statements results in the creation and termination of processes, we can mirror this by dynamically creating and terminating threads at run time.

Multithreading brings with it the issue of thread scheduling. We observe that in regard to concurrent processes, CSP has an *interleaving* model of concurrency [Hinc95] with a loose execution order. For example, consider the following statements:

```
A = p→q→z→SKIP

B = r→s→z→SKIP

C = (A||B)^{z}
```

When process C causes A and B to run concurrently, there is no constraint on the order of A's events relative to B's, except that they must both finish by synchronizing on z. That is, any of these traces would satisfy the specifications: (p,q,r,s,z), (r,s,p,q,z), (p,r,s,q,z), and so on.

This kind of concurrent execution is readily provided by conventional multitasking or multithreading operating systems, regardless of their policies on preemption or priority. In other words, one can use nonpreemptible, equal-priority threads to emulate CSP trace semantics. Or one can use preemptible threads with adjustable priorities, and the required CSP trace semantics will still be maintained. This means that we have a great deal of latitude in our framework's scheduling policy.

The kind of dynamic scheduling needed to accommodate on-the-fly process creation naturally requires more run-time overhead than static scheduling schemes sometimes utilized for hard real-time systems. However, dynamic scheduling is compatible with the requirement for run-time binding of process names (see Section 3.1.4 below), with our method of integrating user code (Section 3.4), and with the general absence of timing constraints in CSP, so there is little motivation to look for alternatives.

3.1.2 Code generation targets

Our purpose is to synthesize code from CSP specifications. In an OO implementation, that means creating objects. Following a typical OO design methodology, we can chose classes directly corresponding to objects in the problem domain, in this case, processes, events (including communication channels), and data items. Processes will need to be so-called *active objects*, i.e., possessing a thread of control.

Armed with these classes, we can map out a strategy for software synthesis. We begin by statically analyzing a set of CSP specifications to identify all object occurrences. A customization file—actually a C++ source program—can then be generated in two sections: (1) compile-time definitions for all passive objects, that is, events and non-trivial data items; and (2) sequential code segments for all processes, packaged as one C++ function per process. Creation of active objects is deferred until run time in order to implement the dynamic occurrence of parallel composition operators (| | and | | |) which, in effect, cause process spawning. These processes, as their execution flows through the function bodies that represent CSP statements, will invoke methods on event objects that cause interprocess synchronization, channel communication, and user code invocation to take place.

3.1.3 Interprocess communication

All interprocess synchronization is required to be carried out via event rendezvous, with channel communication being a special case of event handshaking. This means that concurrent programs specified in CSP do not have a "critical section problem," because there is no concept of shared memory that needs to be protected. Thus, at the level of framework customization, our implementation need not provide any explicit synchronization devices such as semaphores or monitors. All we have to do is replicate CSP's *barrier* [Lewi98] style of multiprocess synchronization via the behaviour of the framework's event objects.

Unfortunately, the straightforward picture of concurrent processes heading toward a synchronization barrier will be complicated by participation in deterministic choice. Our

solution to this problem is too involved to present here—see Section 5.5 on page 94 for details.

If we wish to take advantage of shared memory internally in order to reduce the involvement of the OS in interprocess message passing, we can do so by providing the framework's internal data structures with mutual exclusion locks, utilizing whatever primitives the OS may provide for this purpose. This will be transparent at the level of framework customization, i.e., the level at which software synthesis is taking place.

3.1.4 Binding of symbols

Symbols in CSP are of three categories: (1) process names, (2) event (channel) names, and (3) variable names. Both process names and event names can be subscripted. (In the case of process definitions this is considered as taking arguments.) The binding of the first two categories to compile-time objects is a troublesome area.

Because a process reference can be subscripted by a run-time value, as in `r→S(n)`, and several "S" variants could be defined, a dynamic mechanism is required in order to instantiate the correct version of the process. Note that this represents preparing for a worst-case scenario. Even a subscripted process reference may well be unambiguous in its context and capable of resolution by the translator.

The subscripting of event names similarly requires a runtime mechanism to match up subscript values. That is, if two processes are to synchronize on event `a(2)`, then when, say, `a(n)` occurs, the framework must dynamically evaluate the subscript in order to determine whether to invoke synchronization.

Much more problematic than subscripting is the binding of event names in a pro-

cess's environment. Consider again this simple code sample:

```
A = p→q→z→SKIP

B = r→s→z→SKIP

C = (A||B)^{z}
```

When code is being generated for event z in process A or B, in the case above the translator could actually analyze all three statements and conclude that it needs to generate code to invoke synchronization on z.

However, in the general case, this kind of static analysis is impractical for CSP. In the sample above, if we add the following statement:

```
START = A;B
```

and execution arrives at A and B via process START, then event z *must not* attempt to synchronize the processes. Instead, z will be executed twice.

The difference in these two cases is found in the environments of A and B at run time. A translator cannot deal with this through static analysis alone, unless we are prepared either to restrict the specifications that can be synthesized (the example above, including both C and START, would be rejected), or else perform a complete control flow analysis and generate code for every possible case (i.e., generate a "C" version of A and a "START" version, and the same for B).

CSP's capability of event renaming (# operator) further muddies the water. This is because the translator does not know that z in the process definition will still be "z" at run time; it could be renamed in the process's environment to some other event.

To preserve as far as possible the full flexibility of CSP, even though it means much

more work, we have elected to implement a run-time environment mechanism that allows proper dynamic binding of event names. This elaborate mechanism, called the *environment stack*, is detailed in Section 5.2 on page 87.

3.2 Design goals

In the architectural issues just discussed, solutions have been proposed for all the problems involved in synthesizing C++ software from CSP. It should be noted that some solutions could be redesigned with a view to reducing run-time overhead, most notably the symbol binding mechanism. However, it was judged that at this stage in the research it was more important to preserve flexibility, and that optimization based on different trade-offs could be performed in the future, perhaps in conjunction with transforming CSP++ into a commercial tool.

Now we turn to design goals to be applied in the context of these architectural choices. These are: (1) run-time efficiency, (2) understandable code, and (3) portability. The rationale for these goals and the specific design choices that they engendered will now be discussed.

3.2.1 Run-time efficiency

Having acknowledged earlier that multithreaded C++ may be unattractive altogether for certain highly resource-constrained systems, this is no excuse for eschewing the resource savings that can result from some simple optimizations. With this in mind, a number of design principles were laid down for CSP++:

1. *Favour looping over recursion.*

CSP notation naturally lends itself to a recursive interpretation, as in self-referential processes (e.g., `A(i) ::= c→A(i+1)`) or fixed point (`fix X.` *...expression with* `X`...). While elegant in concept, a conventional stack implementation of recursion can result in an explosion of storage. Both these constructs can just as well be implemented with looping.

2. *Only create processes when control truly forks.*

Similarly, it is tempting to simply create a new process (in the OS sense of thread or task) whenever a process is invoked, but this can result in a rapidly mushrooming environment. Even if 95% of the active processes are only waiting for their descendants to finish, they are still consuming memory, and, depending on process management algorithms, may have to be constantly stepped over in scheduling queues. But much extraneous process creation can be avoided by observing, as in the example above, that when `A(i+1)` starts, there is no need to keep `A(i)` alive; the task running `A(i)` can instead be transformed into `A(i+1)`. This temptation appears again for deterministic choice: Given `a→P | b→Q | c→R`, one might wish to fork three processes to concurrently try events `a`, `b`, and `c`, but this can also be avoided.

3. *Limit storage growth by utilizing automatic (stack) variables and putting heap variables under their control.*

This strategy enlists the C++ compiler to do the storage management and thus avoid the "leakage" to which user-managed schemes are often susceptible.

4. *Avoid dynamic binding features whenever static binding is possible.*

This is a strategy for reducing run-time computation. For example, a dynamic binding solution for identifying agents and actions might be to assign them ID numbers, and then

look them up in a table when they are invoked. However, by giving each a unique external symbol, the loader can be enlisted to bind them to their invocations at load time. In the same spirit, gratuitous use of C++ virtual functions is avoided.

3.2.2 Understandable code

We tried to design the CSP++ translation to be human-readable and similar to CSP. This is not an obvious goal, given that compilers typically produce opaque output (for instance, the C output of AT&T's cfront compiler is extremely cryptic). The purposes are (a) to ease the work of translation (remembering that it had to be done by hand in the initial stages of this work), and (b) to make the generated code readily accessible for checking and debugging. This has been accomplished as follows:

1. relying on C++ operator overloading to create a syntax similar to CSP
2. translating each CSP action as a fresh C++ statement, thus allowing symbolic debuggers (e.g., **gdb**) to set breakpoints, single step execution, etc.

Even when automatic translation is utilized, having understandable high-level output—compared, for example, with assembly statements—makes it convenient to verify that the translator is generating the C++ that one expects, and to debug any associated user-coded procedures.

3.2.3 Portability

Here we speak of the ability to transfer the OOAF to another processor/OS combination different from what it was originally built for. Framework portability has been enhanced by these features:

1. CSP++ is written throughout in ANSI C++; no assembly language has been used.

2. A task library was selected for the initial implementation that was compatible with a variety of architectures, and capable of being layered on top of an OS multithreading scheme.

We succeeded in changing from the AT&T task model under SunOS to POSIX threads under Linux. Details on this port and lessons learned are discussed in Section 3.5 and Section 3.5.3, respectively.

3.3 Class hierarchy

Since this implementation was created for the csp12 dialect in order to conform with the in-house tool mentioned in Chapter 2 above, a slight shift to csp12 terminology needs to be made at this point: CSP processes are known as *agents*, and CSP events are called *actions*. We further distinguish between *channel* actions, which communicate data, and *atomic* actions, which do not. This terminology is reflected in the framework's nomenclature. The most noticeable idiosyncrasy in csp12 statement syntax, relative to that of, say, Hinchey and Jarvis [Hinc95], is that agents are defined by a ": : =" operator rather than the plain equals sign.

The hierarchy of classes used to implement the above strategy is shown in Figure 5, and Figure 6 on page 50, drawn using UML notation [Pool99]. Details of data members and methods are omitted where not necessary for the explanations below. The principal base classes are as follows:

- **Agent**—embodies a process definition, subclassed from AT&T Task Library `task` class, representing a schedulable thread of control

- **Action**—encompasses the two flavors of CSP events, channel actions which pass data, and atomic actions which do not
- **Env**—declarative objects used to introduce an `Action` reference (`ActionRef`) into the environment of a process in one of three roles: for synchronization (`^{a,b,...}`), hiding (`\{a,b,...}`), or renaming (`#{a=b,c=d,...}`)

Each of the above is described in a subsequent section, finishing with the classes devoted to data manipulation.

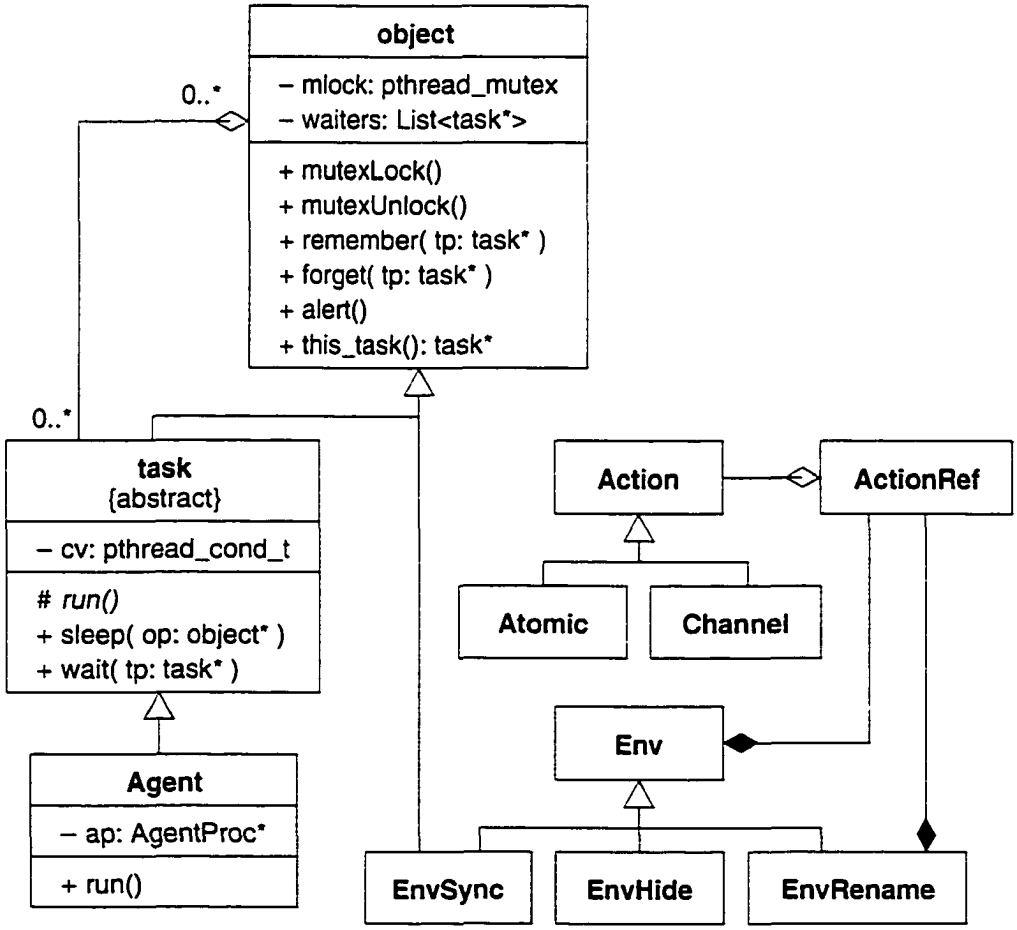


Figure 5: CSP++ non-data class hierarchy

3.3.1 Agent class

Since C++ does not contain the notion of concurrency, we have to provide it ourselves. The AT&T task library is based on a coroutine type of thread model. That is, the user's application runs as a single (heavyweight) Unix process under SunOS, but is free to spawn as many tasks as needed—in our case, csp12 agents. Under this model, any object desiring to have a schedulable thread of control is derived from class `task`. The thread body to be executed is none other than the object's constructor. Normally in C++ the constructor is briefly given control when an object is created. But what being a "task" means is that (a) execution of the constructor will be delayed until the task is dispatched for the first time, and (b) it will thereafter be suspended and resumed according to the operation of scheduling primitives (`task` methods). This is arguably an abuse of the philosophy of C++ constructors, but without concurrency in the language, such contrivances are expected.

This model might suggest that a unique subclass must be created for each agent so that it can have its own constructor. We avoid this class proliferation by making the `Agent` class, which is derived from `task`, a simple function-caller. As was mentioned above, each CSP process definition is translated into an individual function (of type `AgentProc`). An argument to the `Agent` constructor designates which `AgentProc` the task is to run. When that finishes, its return code may designate another `AgentProc` to succeed it. This allows execution to chain from `AgentProc` to `AgentProc` until one ends with the special CSP `SKIP` process, which will terminate that `Agent` task and wake up its waiting parent (also an `Agent` task). The source code in Appendix A provides numerous examples.

Combining inheritance with concurrent synchronization raises the spectre of potential "inheritance anomalies" [Mats93]. However, since our design utilizes a simple form of inheritance in which the `task` base class's synchronization methods are used "as is" (i.e., without being overridden in the `Agent` class, and `Agent` has no subclasses), then according to [Reit97] this pattern of so-called "sequential inheritance" should not engender anomalies.

The above model had to be changed slightly for the LinuxThreads implementation, which is what is depicted in Figure 5 above. The original AT&T task library inserts another class `sched` between `object` and `task`. Only the features of these classes that were actually being used were reimplemented with LinuxThreads, and in the process `sched` was collapsed into `task`. The AT&T `task` constructor appropriated the subclass constructor by "hacking" the caller's stack and, in effect, hijacking the subclass constructor—which in C++ is normally executed after the parent's constructor—for later dispatch as an independent schedulable thread. Without our writing assembly code, this behaviour could not be replicated using LinuxThreads. Instead, the task body (formerly the `Agent` constructor) was moved into a new virtual `Agent::run()` method. This is similar to the technique that Java uses for thread creation [Lea96].

To summarize, whenever the thread of control must fork (as when composing parallel or interleaved agents), one or more new `Agent` objects will be created. Arguments are passed to the generic `Agent::Agent` constructor indicating which `AgentProc` to run and providing its arguments. The parent task would then wait for the newly created subtasks to finish before carrying on.

3.3.2 Action class

The two subclasses, `Atomic` and `Channel`, correspond to the two types of actions available in CSP. Operators are defined to allow function-call syntax to invoke `Atomic` actions (e.g., `foo(2)`), and C++-style I/O syntax to invoke `Channel` actions (e.g., `chan<<1` for `chan!1` output, and `chan>>x` for `chan?x` input). This is similar, though not identical, to CSP syntax, and contributes to CSP++ source code readability.

The above classes are derived from the `Action` base class, which provides common methods needed for executing either kind of action. These lead to searching the agent environment for hiding, renaming, and synchronization orders, performing multi-agent synchronization, handling deterministic choice situations, and printing traces. These operations are explained in Chapter 5.

As was mentioned in Section 2.4 under point 5 on page 32, actions are either intended for internal synchronization use or for linkage to external routines. In the case where no synchronization is ordered, nor has a corresponding external routine been supplied, the `Atomic` and `Channel` classes each exhibit some primitive default behaviour: An `Atomic` action will simply print its name and subscripts, `Channel` output will print its name and value, and `Channel` input will prompt the user to type an integer (or more than one, if a `DatumVar` is the receptacle). These default actions are useful as "stubs" for external routines until they can be written and linked in.

3.3.3 Environment classes

The three subclasses `EnvSync`, `EnvHide`, and `EnvRename` correspond to the three kinds of conditions that can be placed in an agent's execution environment. For example,

```
(A||B)^{c,d}\{c}#{e=f}
```

means:

- A and B will synchronize on actions c and d.
- Hide c (the invoker of this statement will not be aware when c occurs).
- Rename e to f.

This would result in four environment objects being pushed into the environment of parallel composition `(A||B)`: `EnvSync(c)`, `EnvSync(d)`, `EnvHide(c)`, and `EnvRename(d,e)`. The action names themselves would be instances of the `ActionRef` class. Since each environment object contains at least one `ActionRef`, it is stored in the base class `Env`. `EnvRename` contains a second `ActionRef` (here, `e`).

`EnvSync` is the most complex class because it is used to implement synchronization. Since it is subclassed from the `object` class of the task library, it incorporates a list of tasks that are currently waiting on it. That is, an agent needing to wait for synchronization with another agent adds itself to the appropriate `EnvSync` object's wait list by invoking the `task` method `sleep(&theEnvSync)`, which also causes the task to be suspended. In the LinuxThreads version, `sleep()` blocks the task on its own *condition variable*, `cv` [Lewi98].

Later, when the other agent arrives at the synchronizing action, it invokes the method `theEnvSync.alert()`, which wakes up any waiting agents (signals their condition variables). Note that in the AT&T task model, the `task` class inherits from `object`. This is so that a task can be waited for ("joined") by using the same mechanism just described.

Naturally, the manipulation of an `object` instance's `waiters` list must be done

in a critical section. This is ensured by making the rule that callers of `remember()`, `forget()`, `alert()`, `sleep()`, and `wait()` must bracket their use by calling `mutexLock()` and `mutexUnlock()`.

The mutex `object::mlock` is actually a static class variable; that is, there is only one mutex for the system. This may seem unduly heavy-handed, in terms of limiting opportunities for concurrent execution and maximizing contention for the lock, but there is a reason for it. Since deterministic choice in CSP may make an agent party to multiple concurrent synchronization attempts, it is possible for an `Agent` object to put itself on more than one `EnvSync`'s `waiters` list. If each `object` instance had its own mutex, these would all have to be acquired serially, which would open the door to circular waiting and hence to deadlock. By having only a single mutex, deadlock is prevented by denying it a necessary condition [Silb98].

Note that in the AT&T coroutine task library, where preemption was not an issue, no mutexes were required. That apparatus had to be retrofitted for the Pthreads reimplementation.

3.3.4 Data classes

This area presented a challenge in view of the design goals related to storage efficiency (Section 3.2.1), the desire to allow for easy addition of data types in the future, and the determination to make CSP++ data items participate in C++ arithmetic expressions without piling up much special-purpose code. Fortunately, OO technology is very helpful in these areas.

The hierarchy of data classes is shown in Figure 6. These are used to create

`Literal` instances which are passed between agents.The base class for data items is `Literal`. The storage for `Literals` is carefully managed by the *container class* `Lit`. Literals have no public constructors; rather, when a `Lit` is created, it (privately) allocates a new `Literal` of the appropriate subclass on the heap and stores a pointer to it. Each `Literal` keeps a count in `links` of the `Lits` pointing to it. When one `Lit` is assigned to another `Lit`, it is simply a matter of copying `Literal*` pointers and updating link counts. When a link count is reduced to 0, this signifies that no more containers point to the `Literal`, so the `~Literal` destructor is invoked, which releases its storage. This is essentially a garbage collection scheme.

`Lits`, in turn, are fully managed by the C++ compiler as automatic (stack) variables, the principle being that if the compiler manages the `Lits` (as blocks of code go in and out of scope), and the `Lits` manage the Literals (as the `Lits` are created, assigned, and destroyed), no storage leaks should be possible. Furthermore, since `Literals` are heap based, their addresses are valid in any `Agent` task, so there is no difficulty with interchannel data transfers.

`Literals` currently come in two flavours, `Num` (integer value) and `Datum`. The

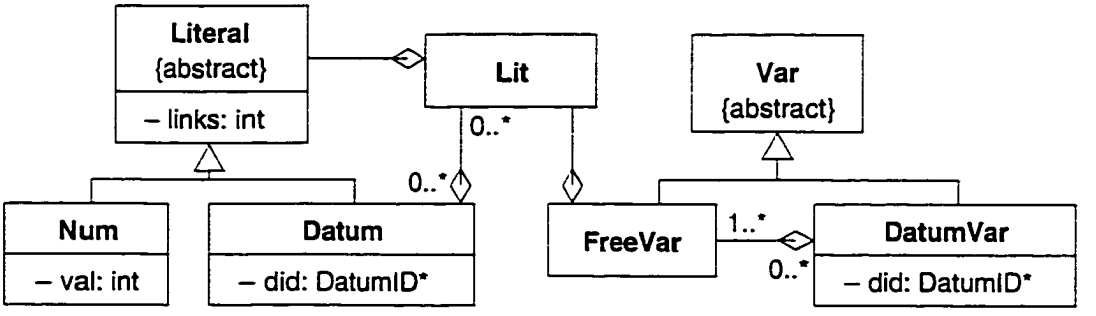


Figure 6: CSP++ data class hierarchy

latter complex data type is the CSP++ equivalent of csp12's **label**, written as a symbolic `DatumID` (`char*`) optionally followed by a series of subscripts in parentheses. Each subscript is a `Lit`, which means it points to any type of `Literal`, possibly another `Datum`. Thus,

```
request( client(10), block(5), flags(buff,eof,10) )
```

would be a valid `Datum` literal. Other `DatumIDs` mentioned here within the `request` `Datum` are `client`, `block`, `flags`, `buff`, and `eof` (the last two having no subscripts).

A `Datum` will be created in a context demanding a `Lit` when the pattern "`DatumID(` *subscripts* `)`" is encountered. For this to work properly, a function with the same name as the `DatumID` is defined to take a certain number of arguments. One such function would be the following:

```
Lit request( Lit a1, Lit a2, Lit a3 ) {...}
```

This function would package up the three subscripts into a `List<Lit>`, and return a `Lit` pointing to the new `Datum`. Part of the CSP++ translator's work is to generate the appropriate `Datum`-constructing function for each label appearing in the CSP specification. One consequence of this approach is that a given label must always be used with the same number of subscripts. This is not felt to be an unreasonable restriction. Other restrictions are listed in Appendix C.

Turning from literals to variables, it is evident that `Lit` containers already function as variables, since they can point to any `Literal`. However, it is desirable to have a special data type corresponding to the concept of "free variables" in CSP notation. These can be required in contexts that demand receptacles, particularly channel input. To this end,

the FreeVar class was defined as containing a Lit* pointer, and DatumVar—by analogy to Datums—a labelled list of FreeVar* pointers. Both are derived from the abstract base class Var, so that either will do in a context requiring a variable.

The DatumVar is troublesome to implement but highly useful. The desired behaviour is as follows (in csp12 notation):

```
P ::= chan!foo(1,2,3)
Q ::= chan?foo(i,j,k)
```

When P and Q synchronize on channel chan, we wish to assign 1 to i, 2 to j, and 3 to k. (The DatumIDs must match: It would be an error for P here to output bar(1).) Thus another function named "foo" is needed for this context, one that will package, not the *values* in FreeVars i, j, and k (as the request function would above, since we can supply a standard conversion from FreeVar to Lit), but rather their FreeVar* addresses. Here is that function:

```
DatumVar foo(FreeVar& a1, FreeVar& a2, FreeVar& a3) {...}
```

Now we can let polymorphism determine which foo datum-builder to invoke—the Datum version or the DatumVar version—depending on the context.

A final note on data classes is that the Literal hierarchy is easily extensible: One need only create additional subclasses of Literal—say, String or Set—and appropriate operators to go with them. All the rest of the data handling mechanisms should take them in stride without any modification.

3.4 Integration of user code

The framework provides "hooks" for linking user code to CSP actions. In the sections following, we first describe how this feature is used in practice, and then give a deeper explanation of the underlying concepts, including participation of user procedures in the CSP choice construct.

3.4.1 Practical overview

Each `Channel` or `Atomic` action can *either* be used for internal synchronization (i.e., with another CSP process) *or* it can be linked to an external routine. Trying to do both with the same action is not permitted and results in a run-time error. In the latter case, when the action is performed, the associated external routine is invoked with arguments corresponding to the `Atomic` subscripts, `Channel` input variables, or `Channel` output values.

This technique of integration is portrayed in Figure 7, which shows how a synthesized CSP program interacts with its external routines. Part (a) shows how the two soft-

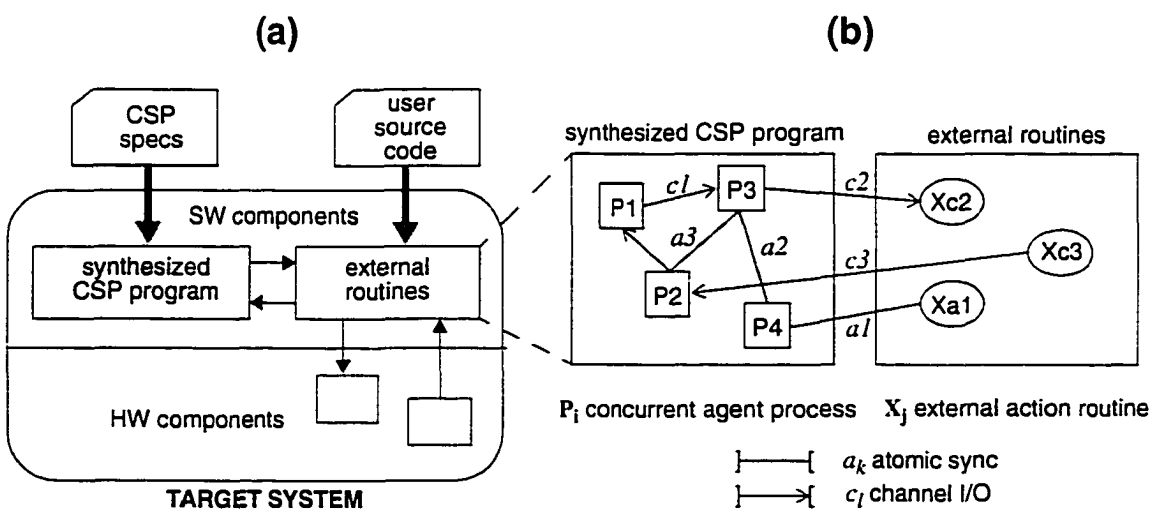


Figure 7: CSP and user code integration

ware components are prepared from the CSP specification and user source code, respectively, and how the latter interacts with the system's hardware components. Part (b) shows processes communicating and synchronizing via internal actions (c1, a2, and a3). Other actions (a1, c2, and c3) are used to invoke the external routines.

In the disk server case study (Section 2.1.2 on page 14), all actions shown in the CSP code are used for internal synchronization. To embed this CSP in a hardware environment, the dummy `Disk` process can simply be removed from the statement that defines `DSS`,

```
DSS = ( (DSched || DQueue)^{enq,deq,next,empty}
      || (DCtrl || Disk)^{dio,dint} )^{dci,dco}
```

leaving the following:

```
DSS = ( (DSched || DQueue)^{enq,deq,next,empty}
      || DCtrl )^{dci,dco}
```

Now the `dio` and `dint` actions are available for external use. When `DCtrl` reaches `dio!_blk`,

```
DCtrl = dci?start(_cl,_blk)→dio!_blk→
                dint→dco!fini(_cl,_blk)→DCtrl
```

the external routine associated with `dio` will be called, which presumably starts the hardware I/O (non-blocking). Similarly, the routine associated with `dint` will block awaiting a completion interrupt (details, of course, being hardware and/or OS dependent). The source code for this example is actually given in Appendix A.4ff.

3.4.2 Conceptual model

In the CSP++ system model, the CSP specification describes the control flow of the system in terms of concurrent processes. The user source code can be regarded as fleshing out the semantics of those actions which interact with other hardware and software components of the system. This model sounds fine from a superficial standpoint, but we need to probe its semantics.

In the disk server example of the preceding section, some indication is given of the steps used in system integration: Initially, the entire system, including agent "stubs" simulating its environment, is described in high-level terms purely through CSP specifications executing actions. Each action is either used for interprocess synchronization and communication, or else it is completely abstract: just a name with no effect except that of printing a trace.

Subsequently, external procedures are introduced and linked with a subset of the actions. This may occur in the context of stripping out the agents that simulated the system environment (e.g., the `Disk` agent above). When that occurs, some actions (like `dio` and `dint`) that were initially used for interprocess synchronization become abstract in the specification. Any abstract action is a candidate for linking with an external action procedure.

Now, the key point to observe is that any action that has *always* been abstract (i.e., never used for synchronization) can be linked with impunity to an external procedure that does anything at all (save, of course, for communicating with another CSP-specified process). Such a procedure can engage in I/O and even block its thread without affecting the

semantics of the model-checked specification. In contrast, an action that has been used for synchronization (both `dio` and `dint` fall into this category), must continue to obey the semantics of synchronization. This should not be surprising, because such actions are in reality being used to synchronize a CSP process with the external system environment. That is to say, `dio` and `dint` were originally synchronizing agent `DCtrl` with dummy agent `Disk`; now they are synchronizing `DCtrl` with the physical disk hardware (or possibly with system calls that perform the I/O instructions). Naturally, the synchronization semantics cannot change.

The place where this model has tricky implications, is in the special demands put on an external procedure when its associated CSP action is participating in a deterministic choice construct. As will be shown in detail in Section 5.5, a choice statement such as,

```
a→E | b→F
```

means that action `a` has to be tried first, and if it doesn't succeed, then `b` is tried. If neither succeeds, the run-time mechanism lies in wait for a future, delayed success from either action. This implies that an external procedure which is going to participate in choice must be capable of being "tried," of setting up an asynchronous call-back that informs the run-time mechanism of a delayed success, and of being cancelled when another choice succeeds first.

The above description hints that the interface between CSP++ and the external action procedures is not trivial. On the other hand, neither are such requirements unprecedented: After all, the OS uses just such a regime whenever it sets up asynchronous I/O, gets a call-back from an interrupt handler, and cancels an I/O request due to a timeout.

In the current version of CSP++, only the most fundamental part of this interface has been implemented. For actions that do not take part in choice, this limitation is of no practical consequence. For actions that *do* take part in choice, the effect of the current limited implementation is that an action is deemed to succeed as soon as its external procedure returns to the framework.

Nonetheless, one can still set up a choice situation with external procedures by embedding their actions in concurrent agents. For example, suppose one wants to write the following agent:

```
P ::= xa→Q | xb→R
```

where `xa` and `xb` are linked with external procedures. In the current version of CSP++, control will stall on `xa` until it returns, then pass to agent `Q`, and `xb` will never be tried. But this variation should have the desired effect:

```
Pa ::= xa→dida→SKIP
Pb ::= xb→didb→SKIP
Pc ::= dida→Q | didb→R
P ::= ( Pc || ( Pa|||Pb) )^{dida,didb}
```

Additional internal actions, `dida` and `didb`, have been created to report the occurrence of external actions `xa` and `xb`, and agent `Pc` has been set up to synchronize the choice.

Completion of the external procedure interface, with full attention to deterministic choice, is listed for future work (see Section 7.2.2).

3.5 Platforms

The first version of CSP++ was created on the SunOS platform that was readily available at the time. In preparing this research for publication [Gard99b], which entailed distributing the source code, it was felt that readers would better appreciate code they had a reasonable chance of running, so to that end a more common platform was targeted. Furthermore, the exercise of porting would offer an excellent opportunity to find out whether the OOAF approach would prove portable or not. These two platforms are described below.

3.5.1 AT&T cfront on SunOS with USL coroutines

Working on a Sun 4 (SunOS Version 4 Unix), we wanted a C++ compiler that came with the following features: (1) some implementation of multithreading or multitasking; and (2) a class library with template support for basic data structures (linked list, bit vector, and so on). These needs were filled by AT&T C++ Version 3.02 (cfront, an early C++ compiler that emitted C code) and its associated USL Standard Components Library (SC-3.0). This library includes an OO multitasking model.

Things were not quite this simple though, because it turned out that the USL task library ironically did not support the Sun 4 architecture that we were compiling on. Fortunately, the necessary object files were found in the Sun C++ distribution (SC1.0, in reality an older version of cfront, V2.1), although that compiler had to be rejected due to its inability to handle templates.

Thus, V1.0 of CSP++ was built using the task.h header (slightly modified to fix an external symbol problem) and libtask.a from Sun's SC1.0 library, the rest of the USL SC-

3.0 library (apart from task.h), and the AT&T C++ compiler. This version includes the full OOAF, but not the CSP-to-C++ translator, since at that time translation was still being carried out by hand.

3.5.2 GNU g++ on Red Hat Linux with Pthreads

By 1999, AT&T cfront was essentially obsolete and its Standard Components library had been superseded by the C++ Standard Template Library or STL [Aust99]. We decided to attempt a port of the V1.0 C++ code to GNU g++ (ecgs version 2.90.29) on Red Hat Linux 5.2, a popular x86 version of Linux. g++ includes the STL, which we used in the translator, cspt (see Chapter 6).

But what were we to do with V1.0's dependency on the USL Standard Components and task libraries? Mindful of the programmer's maxim, "Never change code that works," it was considered that substituting STL templates into the existing framework code was risky. Therefore, we simply copied the few SC-3.0 files that we needed and recompiled them with g++. This proved entirely effective, if not wholly satisfactory from a "house-keeping" standpoint.

However, this simple approach—recompiling with g++ for Linux—could not work for the task library since it would have meant porting assembly language from the Sun to the PC. The AT&T task model uses assembly language to modify the stack whenever a subclass constructor inheriting from class `task` is invoked (see Section 3.3.1 on page 45). Assembly language is used again for swapping stacks during task rescheduling. We were determined not to use assembly code due to its adverse impact on portability.

Instead, we opted to switch to the POSIX threads (Pthreads) model, LinuxThreads

[LXT], that came with Red Hat Linux. The `task` base class was reimplemented using calls to Pthreads routines, and the framework code which inherits from `task` was very largely unchanged.

AT&T tasks are actually *coroutines*. That is, they have no concept of priorities or preemption—a task runs until it voluntarily yields control to another task. This is known as the "many-to-one" model, meaning that many threads are swapped on and off of one kernel-scheduled entity (this could be a single heavyweight Unix process in SunOS, for example). That is to say, preemption of a coroutine by another kernel-scheduled entity can occur, and in Unix does so constantly (e.g., servicing interrupts), but this is of no consequence to the set of coroutines (aside from timing consequences, naturally).

POSIX threads can be implemented as coroutines, but need not be [Lewi98]. The LinuxThreads version of Pthreads is built on lightweight kernel-scheduled threads, which includes the possibility of thread preemption. This is known as the "one-to-one" model.

Changing from coroutines to preemptible threads could not be totally transparent with respect to the framework code's inherited classes; though, interestingly, the reverse would have been so. This is because preemptible threads represents the more general case, in regards to requirements for locking resources that are shared among multiple threads.

The Pthreads version of CSP++ is publicly available as noted in Section 7.3 (Table 4 on page 116). This version can also be compiled and run on Solaris for the Sun SPARC architecture without modification.

3.5.3 Lessons from Linux port

To fairly evaluate the framework's degree of portability in the rather extreme circumstance of changing the task model, we should first clarify expectations. In order of most portable to least portable, the following conditions would obtain:

1. Totally portability: no source code changes required whatsoever.
2. Code changes in the task library (classes `task` and `object`), but not in the framework classes.
3. Code changes in the framework classes, but not in customization (expressed as the translation algorithms of Chapter 4).
4. Code changes throughout.

It should go without saying that "changes" here does not refer to the dubious practice of peppering the source code with OS-dependent `#ifdef` statements. That sort of illusory "portability" is not what we had in mind.

Level 1 was never expected, so level 2 would have been ideal. The task library was "fair game" for any kind of changes as it originated in the USL library as a self-contained scheduler for a threadless OS, and was being reengineered as a thin interface to POSIX threads.

An outline of the steps taken in porting CSP++ from AT&T coroutines to LinuxThreads has already been presented in Sections 3.3.1, 3.3.3, and 3.5.2. In short, the introduction of thread preemption made it necessary to institute a locking discipline for shared data structures. As far as possible, these changes were absorbed within the task and object classes (level 2), but some slight changes to the Agent and Action classes were unavoidable. In any event, no changes were necessitated to the translation algorithms,

which is the main thing if affording a significant degree of portability.

It could be argued that the locking of shared data structures is de rigueur for concurrent programming and should have been part of the original version. If we accepted this argument, then the changes above could be considered a remediation of original deficiencies, and portability could then be assessed at or near level 2.

However, the argument is simplistic: First, coroutines provide a sort of security blanket, or as Lewis calls it, a "superior programming paradigm" [Lewi98:62], because one does not have to worry about the consequences of something that cannot occur, namely preemption. Second, the point is moot in any case, because the AT&T task model provided no locking mechanisms (no mutexes, no semaphores, no monitors), nor any primitives from which they could be constructed. There was never any way to incorporate these features in the original design of CSP++, working within the AT&T task model.

The above comments might produce the impression that switching from the coroutines model to kernel-scheduled threads should be regarded a necessary evil. That is not at all the case. Quite to the contrary, the latter model gives external procedures the freedom to block with impunity, say for I/O, without causing scheduling of the entire CSP++ system to freeze. This is a major advantage over the many-to-one scheduling model.

Having achieved success with the Linux port, we tried to recompile the new version, still using g++, on a Sun workstation under Solaris, which also supports Pthreads. This turned out to be a true "level 1" port, and worked immediately. Ironically, that ready victory reveals less about CSP++'s portability than the more hard-won struggle to change the task model. Maybe the Solaris port was "too easy" and would have worked whether

OOAF technology was used or not. In contrast, the LinuxThreads port revealed the boundary between OS-dependent and OS-independent code, and as hoped, the boundary fell, to an overwhelming extent, at the border between the framework classes and the task library.

This chapter has described the basic ingredients of the CSP++ framework, which are sufficient to build executable and extensible specifications. The next chapter details how each construct in CSP can be translated into C++ code that creates or invokes these framework elements.

CHAPTER 4

Representation of CSP Statements in CSP++

Given the CSP++ framework described in the previous chapter, customizing it for a particular CSP specification is a matter of systematically translating each csp12 construct into C++ code that invokes the elements of the OOAF. The method of doing this is one of the chief results of this research.

Chapter 4 states the rules for this translation, which can be manually or automatically carried out. Naturally, manual translation is tedious and error-prone, so an automatic translation tool forms an indispensable part of the CSP++ design flow. This translator, called **cspt**, is sketched in this chapter, with the full design details appearing in Appendix D.

Some csp12 constructs have not been implemented yet, due to some ambiguities of interpretation that will be explained, and some only partially implemented. These are listed at the end.

4.1 Naming conventions

Most names—variables, actions, and datums—can be copied directly from their csp12 form; however, agents need more elaborate conventions because of multiple definitions. The issue of run-time binding also arises for agents having variable arguments. Finally, generated names are needed as handles on actions that will be pushed on the environment stack to implement synchronization, hiding, and renaming.

A comprehensive chart of all translator-generated names is given in Table 1.

Pattern	Usage	Section References
Where π = agent name:		
π_*sig*	AgentProc name; *sig* = argument signature	4.1.1
π_*sig*_v	Globalized FreeVar instance; v = variable name	4.3.5
AG_π_*sig*	AgentID instance	4.2.2
π_s*n*	AgentProc name of *n*th extracted subagent	4.1.1, 4.3.5
π_x	Array of ints: [arg index, value] descriptor pairs	4.1.2
π_y	Array of AgentProc*	4.1.2
π_b	AgentBinder instance	4.1.2
Where κ = action name:		
κ_r	ActionRef instance for Action κ	4.1.3
κ_r_*subs*	ActionRef for subscripted Atomic κ(*subs*)	4.1.3
κ_p	Preprocessor symbol, name of ActionProc	4.2.2
Where δ = DatumVar name:		
δ_dv	DatumVar temporary for Channel input	4.3.9

Table 1: Translator-generated names

4.1.1 AgentProc signatures

Consider, a CSP specification containing the following agent definitions:

```
P(0,i) ::= ...

P(1,i) ::= ...
```

These two must be translated into separate C++ procedures, called AgentProcs, so they need different names.

AgentProcs will be named according to their argument *signature*. The two above would be P_c0v and P_c1v, where "c" indicates a constant argument (with its value following) and "v" a variable argument. This allows the translator to set up the right invocation depending on the arguments used. The empty string stands for a nil argument list.

A special signature of s1, s2, etc., is affixed for subagents that are created from

extracted subexpressions (see Section 4.2.3 below). Subagents are never subject to run-time binding.

4.1.2 Agent binders

Static binding is the preferred way to match arguments with the appropriate signature. However, in the case of agents with *constant* argument definitions being invoked with *variable* arguments, run-time binding is required. That is, which signature above matches the invocation of `P` below?

```
Q ::= chan?n→P(n,0)
```

This depends on the value of `n`, which can only be determined at run time.

Agents requiring run-time binding must supply a table of instructions for the binder to consult. A detailed explanation of binding tables is given in Section 5.1 on page 85, but in general the contents are pairs of numbers $[i, v]$, where i is the index of a constant argument, and v is its integer value. The two `P` definitions above would be described by [0, 0], meaning "$arg_0 = 0$", and [0, 1], "$arg_0 = 1$". In addition to a table of argument descriptors, another table of `AgentProc*` pointers is supplied. Then if a descriptor in the jth set of pairs matches the given arguments, the jth `AgentProc` will be called.

The pair of tables described above are declared as follows:

```
static int agent_x[] = { descriptor pairs };
static AgentProc* agent_y[] = { AgentProcs };
```

Next an `AgentBinder` object is declared in order to associate the tables with an agent name:

```
static AgentBinder
```

```
agent_b( agent_x, agent_y, "agent", num args );
```

In the example above, these items would be named `P_x`, `P_y`, and `P_b`.

4.1.3 Action references

`ActionRef` (Action.h) objects are used to prepare `Action` names for pushing on the environment stack. An `ActionRef` for an Atomic must give the number of subscripts followed by their (integer) values. The names of these objects are generated in global scope since action names are themselves global. There can be more `ActionRefs` than `Actions`, because differently subscripted references to the same `Atomic` name are treated as referring to different actions.

4.2 Translated source code

The cspt translator (see Appendix D) operates in two phases: first, a combined lexical and syntax phase which scans the csp12 input file and produces a syntax tree; second, a code generation phase that walks the tree and produces a C++ output file. In addition to the syntax tree, the other data structures that persist between phases are the symbol tables.

The C++ source file resulting from translation has four sections: header file inclusions, global declarations, translated agent bodies, and finally the main program. These are described next.

4.2.1 Header files

This section calls up the C++ ".h" files for the CSP++ class definitions. They are Lit.h, Agent.h, Action.h, and main.h. In the LinuxThreads version, we also include Listio.c and List.c from the USL Standard Component Library, as explained in Note 1 on Page 138.

4.2.2 Declarations

In order to accommodate forward references within agent bodies, all non-local symbols must be collected by the translator and emitted here. Another purpose is to associate compile-time names with ASCII strings, so that run-time diagnostics can print the names of the offending agents, actions, and datums. These declarations fall into three categories:

1. *Agent definitions:* The `AGENTDEF` macro (Agent.h) is used to declare each AgentProc signature. Returning to the example in Section 4.1.1 above, the two definitions of P would be generated as follows:

```
AGENTDEF(P_c0v, "P", 1);
AGENTDEF(P_c1v, "P", 1);
```

   where 1 here specifies the number of arguments. `AGENTDEF` generates a declaration of the `AgentProc`, for forward referencing purposes. The macro also secretly defines a symbol of the form `AG_`*signature*, of type `AgentID`, to point to the agent's ASCII name.

2. *Action definitions:* The class names `Atomic` and `Channel` (Action.h) are used as in these examples:

```
Atomic nak("nak");
Atomic ack("ack", 1);
Channel next("next");
```

   where 1 specifies that the `Atomic` has that number of subscripts (e.g., `ack(5)`). If an external routine is being linked with a CSP action, it needs to be declared as an external of type `ActionProc`, and inserted in the action definition:

```
extern ActionProc buttonProc;
Atomic button("button", 0, buttonProc);
```

   To avoid having to hardcode routine names like `buttonProc` into CSP specifications, the translator substitutes a preprocessor symbol of the form *action*`_p`. If the symbol is defined at compile time, its value is used for the `ActionProc` name; otherwise, no routine is linked in.

3. *Datum definitions:* The DATUMDEF macros (Lit.h) are used to declare each DatumID:

```
DATUMDEF(foo, 0);
DATUMDEF(bar, 1);
```

where, again, the number specifies the subscripts. Then one could later write in an agent body, foo (no subscripts) or bar(0), bar(foo), bar(bar(10)), and so on.

4.2.3 Agent bodies

Each CSP agent body must be translated into one or more AgentProcs, depending on whether or not complex subexpressions are present. For convenience, the AGENTPROC macro (Agent.h) is provided, so we have:

```
AGENTPROC( P_c0v )
   ... translated body goes here ...
}
```

This macro includes code to establish the agent's identity as "P" (by referencing its AgentID) for the sake of any diagnostics that may print out while it is executing.

4.2.4 Main program

The last section is the standard C++ main program:

```
main( int argc, char* argv[] )
{
   MAIN( argc, argv, top );
}
```

The MAIN macro processes any command line options (see Appendix B.4) and creates an Agent task to run the top-level AgentProc *top*.

One reason that "main" is coded here within the translated source is because when the program is executed in a graphical debugger (e.g., **xxgdb** or **ddd**), main's source file is automatically displayed. The agent body code will thus appear in front of the programmer, which is convenient for setting breakpoints in the translated code.

4.3 Agent body translations

In the following code samples, the shaded column is CSP notation (csp12) and the right column is the corresponding CSP++.

4.3.1 Agent arguments and free variables

Variable names in csp12 start with an underscore. C++ can accept this, so the names may be copied directly.

```
P(_x) ::= chan?_y→foo(_x+_y) ...
```

```
#define _x ARG(0)
FreeVar _y;
  //rest of body
#undef _x
```

In this example, _x is the 0th argument of agent P. The `#define` allows the agent body to refer to it by its name _x. The macro ARG (Agent.h) generates code to reference the 0th `Lit` in the Agent's argument array. A complementary `#undef` is needed to avoid interfering with any other uses of the symbol _x later in the program.

This CSP statement has a single free variable. All free variables should be declared with class `FreeVar` at the start of the block.

4.3.2 Prefix

This basic invocation of actions comes in two flavours, `Atomic` and `Channel`.

```
P(_x) ::= chan?_y→foo(_x+_y)→a ...
```

```
#define _x ARG(0)
FreeVar _y;

char>>_y;
foo(_x+_y);
a();
//rest of body
```

Each prefix action becomes a separate C++ statement. ">>" is input, "<<" is output. Variables in expressions are automatically converted to integers due to the provision of a virtual `int()` operator in the `Literal` base class. (There will be a runtime error if the `Literal` type is not Num). Unsubscripted `Atomics` take an empty function argument list.

4.3.3 Environment stack

Action names for hiding, synchronizing, and renaming need to be pushed onto the Agent's environment stack (more about this stack in the chapter on run-time operation, Section 5.2 on page 87).

```
Q#{foo=bar}
```

```
static ActionRef foo_r(foo),
                 bar_r(bar);
foo_r.rename( bar_r );
  //invoke Q
Agent::popEnv( 1 );
```

```
((S||T)
  ^{a(6),b})\{b}
```

```
static ActionRef a_r_6(a(1,6)),
                 b_r(b);
b_r.hide();
a_r_6.sync();
b_r.sync();
  //invoke S||T
Agent::popEnv( 3 );
```

As explained in Section 4.1.3 above, `ActionRefs` appear in the global declara-

tions. These objects encapsulate a reference to a particular `Channel` or (possibly subscripted) `Atomic` action.

Three classes which are not seen above, `EnvHide`, `EnvSync`, and `EnvRename` (Agent.h), are instantiated and linked onto the stack of the current `Agent` when the `hide()`, `sync()`, and `rename(ActionRef)` methods of `ActionRef` are invoked. An inverse method, `Agent::popEnv(`*n*`)`, removes and deletes a specified number of objects. Note that popping is not needed at the logical end of an agent body (see next section).

4.3.4 Agent constants

Agent bodies often end with an agent constant. In this case, the agent can *chain* to the next one, which causes the resources of the present task to be reused.

```
P(_x) ::= Q(_x)#{foo=bar}     #define _x ARG(0)
                              //push Q's environment
                              CHAIN1( Q_, _x );
-------------------------------------------------------
Z(_x) ::= a→L(1,_x)           #define _x ARG(1)
                              a();
                              CHAIN2( L_b, 1, _x );
```

The `CHAIN`*n* macro (Agent.h) hands over control to another `AgentProc`, after reusing its own argument array to store *n* arguments.

Why needn't `Q`'s environment be explicitly popped in this case? The answer is that `Q` will either chain to another agent—in which case the renaming `{foo=bar}` should still apply to its dynamic descendants—or else it will exit with `SKIP`. That will terminate the `Agent` task, cleaning up its branch of the environment stack in the process.

In the first example above, we assume the translator is able to statically determine

which AgentProc to invoke. But for the second, suppose three L agents are known:

```
L(1, 1) ::= ...
L(1, 2) ::= ...
L(2, _i) ::= ...
```

Their AgentProc signatures would be L_c1c1, L_c1c2, and L_c2v. Since the value of _x is not available at compile time, run-time binding is needed. Thus the translator codes CHAIN2(L_b, ...), referencing the AgentBinder object defined for L instead of one of the three fixed AgentProcs. The CHAIN2 macro invokes bind(*args*) on L_b to obtain the correct AgentProc, and then the transfer of control proceeds in the usual way.

4.3.5 Composition (parallel, subordination, and interleaving)

In csp12, parallel composition (|[ ]|) requires an explicit synchronization list, in that the tool makes no attempt to infer the agents' common alphabet. This is also true of subordination (P//Q), where a sync list equal to αP (alphabet of P) must be supplied. Similarly, interleaving (|||) is just composition with no synchronization list. Thus, it suffices to handle the general parallel case.

```
R ::= ((S||T(2))                    //push environment
        ^{a(6),b})\{b}              Agent::compose( 2 );
                                    Agent* a1=START0( S_, 0 );
                                    Agent* a2=START1( T_, 1, 2 );
                                    WAIT(a1);
                                    WAIT(a2);
```

The Agent::compose(*m*) method prepares an *m*-way process fork. Each START*n* macro (Agent.h) creates a new Agent task, and specifies the AgentProc signature (with *n* arguments) that it should start running. The second argument is a *branch*

number (from 0 to *m*–1) to associate with that new `Agent`. It will apply to any synchronization attempts originating in that branch of the dynamic process tree. The returned `Agent*` pointer allows the `WAIT` macro to suspend the present task (by invoking *this-agent*`.wait(Agent*) )` until the descendant terminates. `WAIT` also deletes the Agent upon wake up.

Things become more complicated when *expressions* are composed, rather than simple agent names:

```
R(_x) ::=                    AGENTPROC( R_s1 )
   (a→P(_x)||b→Q)^{c}        #define _x ARG(0)
                               a();
                               CHAIN1( P_v, _x );
                             #undef _x
                             }
                             AGENTPROC( R_s2 )
                               b();
                               CHAIN0( Q_ );
                             }
                             AGENTPROC( R_v )
                             #define _x ARG(0)
                               //push environment
                             Agent::compose( 2 );
                             Agent* a1=START1( R_s1, 0, _x );
                             Agent* a2=START0( R_s2, 1 );
                             WAIT(a1);
                             WAIT(a2);
                             #undef _x
                             }
```

In such cases, the expressions need to be extracted as *subagents* and translated into separate `AgentProcs`. (We use the signature "_*sn*" to indicate translator-extracted subagents.) Care must be taken to ensure that such subagents have access to the arguments and variables of the parent. Part of this can be achieved by passing the parent's arguments (in this case, _x) to the subagent. Variables referenced in subagents are dealt with by promoting them to global scope and generating unique names (prefixed with the par-

ent's signature) to avoid aliasing.

4.3.6 Sequential composition

In sequential composition it is necessary for the parent Agent to stay alive while all but the final Agent are executing:

```
R ::= S#{foo=bar}; T        //push S's environment
                            Agent::compose( 1 );
                            Agent* a1=START0( S_, 0 );
                            WAIT(a1);
                            Agent::popEnv( 1 );
                            CHAIN0( T_ );
```

In this example, S's environment must be popped before chaining to T.

4.3.7 Loop

Section 4.3.4 dealt with the case where agent bodies end with an agent constant. If that constant is preceded by the "loop" operator (@), it must be treated differently.

```
P ::= a?_x→@Q(_x)       a>>_x;
                        while(1) {
                          Agent::compose( 1 );
                          Agent* a1=START1( Q_v, 0, _x);
                          WAIT(a1);
                        }
```

Here the loop is translated as an infinite "while" loop. Each time a new Q agent terminates, the loop is recycled and another Q created.

This implementation is not particularly efficient. It would be preferable to have Q do its own looping, since that would not involve the wasteful creation/deletion of Agent tasks. This could be accomplished by passing a "loop flag" to the Agent constructor. If the flag is found to be set, the Agent must save its initial AgentProc (here Q_v) and

argument array (including _x) so that it can restart execution when its body finishes.

4.3.8 Fixed point

This is a shorthand csp12 way of putting an agent body in-line. The simple case is where the fixed name comes at the logical end of the expression:

```
R ::= a→                    a();
  fix X.(c→X | b→S)         while(1) {
                              Agent::startDChoice(2);
                                 c();
                                 b();
                              switch(Agent::whichDChoice())
                              {
                                 case 0: continue;
                                 case 1: CHAIN0(S_);
                              }
                            }
```

A fixed point expression can be handled as an infinite "while" loop (as opposed to recursion). Reinvocations of the fixed name are treated as escaping back to the loop (so such constructs should specify a means of termination, done in this example by deterministic choice).

It is obvious that the translation above is inadequate for expressions where the fixed name is embedded. An example (not necessarily useful) would be:

```
R ::= a→fix X.( b→S | c→(X|||Y) | d→(X;Z) )
```

These invocations of X need to be handled by new agents. In such cases, the translator should extract the expression as a subagent (say an `AgentProc` called `R_s1` in this case) so it can be STARTed. This would result in the following:

```
R ::= a→R_s1

R_s1 ::= b→S | c→(R_s1|||Y) | d→(R_s1;Z)
```

As with subagents extracted for composition (Section 4.3.5), access to the parent's arguments and variables must be provided.

4.3.9 Deterministic choice

Such choices are specified in terms of several alternative prefixes. The choice whose initial action succeeds first is taken, and the alternatives are abandoned.

```
P ::= a?_x→Q(_x) |
  b→d→R | c!foo→S
```

```
Agent::startDChoice( 3 );
  a>>_x;
  b();
  c<<foo;
switch( Agent::whichDChoice() )
{
  case 0: CHAIN1( Q_v, _x );
  case 1:
     d();
     CHAIN0( R_ );
  case 2: CHAIN0( S_ );
}
```

This is the most elaborate construct in CSP++. The `Agent::startDChoice(`*n*`)` method prepares for an *n*-way choice. The initial actions of the *n* prefixes are tried in turn until one succeeds, either because synchronization with another agent was achieved on that action, or, for external actions, because the external routine returned a success code. When `Agent::whichDChoice()` is invoked, it checks to see whether any of the preceding actions succeeded. If not, it suspends the `Agent` until some success is signalled. (The signalling `Agent` will also have cancelled any outstanding actions which were still waiting.) Finally, it returns a choice number and the switch statement selects the corresponding case.

Note the implications of the above behaviour on the channel input construct translated into C++ as `a>>_x`. If this statement were being executed *outside* the context of

deterministic choice, the call to the extraction (>>) operator would not return until synchronization and data transfer had completed. In contrast, inside a choice construct the operator returns immediately. This can be a problem, because any temporaries which the C++ compiler generates on the stack will go out of scope as soon as the statement is finished.

In practice, this only causes trouble for channel input into a `DatumVar`, for example, `a>>foo(_x)` instead of `a>>_x`. Recall that `foo` here is actually a function call that returns a `DatumVar` (see Section 3.3.4 on page 49). The stack temporary generated for `foo`'s return value can be captured in block scope by declaring a `DatumVar` explicitly, thus:

```
Agent::startDChoice( 3 );
   DatumVar foo_dv = foo(_x);
   a>>foo_dv;
   ...
```

In this example, `foo_dv` is an explicit translator temporary which will not go out of scope until the surrounding block terminates, by which time the execution of `whichDChoice()` will have ensured that the data transfer (here, into variable `_x`) is complete.

4.3.10 Conditionals

Conditional expressions can be translated directly into C++, taking care to substitute the latter's version of comparison operators, csp12's syntax being slight different:

```
if _x = 1 then P          if (_x == 1) {
                            CHAIN0( P_ );
                          }
```

Also see Section 4.4.1 on page 80 concerning the use of conditionals with general choice.

4.3.11 Agent termination

Agents may explicitly terminate in any of three ways:

1. By chaining to another `AgentProc` (discussed above in Section 4.3.4).
2. By executing `SKIP`. This is translated as the `END_AGENT` macro (Agent.h), which simply returns zero. This value terminates the `Agent` task, which will wake up a waiting parent.
3. By executing `STOP`. This is translated as a call to `Agent::stop()`. A dump of all active `Agent` tasks is printed, and then the program exits to the operating system.

Otherwise, an agent's last act will be to start one or more new `Agents`. In that case, implicit termination, as if `SKIP` had been executed, will occur when those descendant `Agents` themselves terminate.

4.3.12 Arithmetic expressions

csp12 operators must be translated to their C++ equivalents, as shown in the chart below:

csp12	C++
+ - *	*same*
= <>	== !=
< <= > >=	*same*
mod	%
not available	/

Division does not appear in the csp12 syntax definition (this may simply be a typographical error), but if the standard "/" operator is coded in a specification, it can be easily mapped to the C++ division operator.

4.3.13 Pipe

This derived operator (▷ or |>) is implemented as in csp12, that is, by expansion to primitive operators. That is, P |> Q expands to:

```
(((P#{right=comm})||Q#{left=comm}))^{comm})\{comm}
```

Agents being "pipelined" must have `left` and `right` channels for this to be effective. P's `right` channel is synchronized with Q's `left` channel by renaming them both to comm, which is then hidden from the enclosing environment.

4.4 Partially implemented constructs

The implementation listed in this section is not considered complete. Its nondeterminism is problematic and more study is needed. Generally speaking, nondeterminism can be useful for keeping specifications at a high level of abstraction, but its semantics from the standpoint of code generation are open to debate.

4.4.1 General choice

Also called nondeterministic choice, a prominent use is in conjunction with conditional expressions.

```
R(_x) ::= (if _x = 1 then P) + Q    if (_x == 1) {
                                      CHAIN0( P_ );
                                    }
                                    CHAIN0( Q_ );
```

Aside from this usage, the desired semantics seems to be a kind of extended deterministic choice. Consider the following sample:

```
P ::= a→(Q + R + S)
```

[Chen94] explains that Q, R, and S may either have a common initial action, or they may not. If not, we will choose among them based on the agent whose initial action occurs first. In effect, we wish to wait until one of the agents Q, R, or S has gotten started, and then abandon the other two. This could be handled very similarly to deterministic choice. Instead of waiting locally for the first of several actions to complete, we can start three `Agents`, also setting flags so that the first successful action in *any* of them will cause the other two to be cancelled. This presents a messier unwinding problem than deterministic choice, but it should still be possible to implement.

On the other hand, if Q, R, and S *do* have a common initial action, we should choose an agent "nondeterministically." In practice, we could still follow the behaviour described above. Then what will actually happen is that the first agent listed in the "+" expression will have priority. This may be a satisfactory implementation, though it is not truly "nondeterministic."

4.5 Future constructs

Constructs listed in this section caused difficulties. Since they are special features of the csp12 dialect, they are left for future implementation.

4.5.1 Menu

This construct is employed in the following sample:

```
_x:{a,b,c}→P(_x)
```

The idea is that some action in the menu `{a,b,c}` must be taken. Its identity is recorded in variable _x, which value may be used in succeeding agents or expressions. One appli-

cation is menus of *numerically-named* actions—e.g., a list of coin denominations {1,5,10,25} inserted into a vending machine—the intention being to operate on the number in a subsequent arithmetic expression, or, as in the sample above, to use its value as an argument or subscript.

CSP++ does not currently allow numerically-named actions (they would not be legal C++ identifiers). If this is all that is desired, we could define a new subclass of `Atomic`, say `NumAtomic`. The translator could deal with the sample above as follows:

```
NumAtomic na_1(1), na_5(5), na_10(10), na_25(25);
FreeVar _x;
Agent::startDChoice( 4 );
   na_1();
   na_5();
   na_10();
   na_25();
switch ( Agent::whichDChoice() ) {
   case 0: _x=na_1; break;
   case 1: _x=na_5; break;
   case 2: _x=na_10; break;
   case 3: _x=na_25; break;
}
```

For this to work, there would need to be defined a `NumAtomic` to `Lit` conversion that produces the numeric value coded in the `NumAtomic` declaration.

If something more elaborate than numerically-named actions is wanted, then the most useful semantics is not obvious. Therefore this construct is left for further study and future implementation.

4.5.2 Independent actions

This is another way of combining actions using "&" that is defined in csp12:

```
(a & b & c)→P
```

The meaning is that all the actions—a, b, and c—must be taken independently, i.e., , in no particular order, before going on to P. This is similar to the prefix expression,

```
a→b→c→P
```

except that in the latter the order is defined. Two ways of implementing independence in CSP++ are the following:

1. Extract the actions as `Agent` subexpressions and translate as interleaving:

   ```
   ( a→SKIP ||| b→SKIP ||| c→SKIP )→P
   ```

2. Handle similarly to deterministic choice, but instead of having the first-occurring action cancel the rest, waiting would continue until all had completed. For example:

   ```
   Agent::startIndep( 3 );
      a();
      b();
      c();
   Agent::waitIndep();
   CHAIN0( P_ );
   ```

The first approach is crude (in that it involves starting tasks that are not truly required), but it could be done within existing CSP++. The latter approach, essentially a multiprocess synchronization technique known as a "barrier" [Lewi98], is probably preferable.

Now that the method of statically translating CSP constructs into OOAF terms has

been explained, we are ready to describe in more detail the dynamic operation of the framework's components at run time. This is the subject of the next chapter.

CHAPTER 5

CSP++ Run-time Operation

Execution of a C++ program commences at the `main()` function, which for CSP++ handles any command line arguments (see Appendix B.4) and then simply creates the `SYS Agent`. Depending on how the user has specified `SYS`, it will in turn spawn other `Agents` and these will invoke actions, including their associated external routines.

To fully appreciate how the framework objects—chiefly `Agent` and `Action` instances—collaborate in executing a translated specification, several areas need to be elucidated:

- the algorithm used for run-time binding of variable-argument agent invocations to the proper one of several candidate definitions
- the run-time stack structure that enables action execution to take place in a multi-layered environment of synchronization lists, action renaming, and concealment
- the means of synchronizing multiple concurrent agents on a common action
- the mechanism used to implement deterministic choice

These vital operations, which form the heart of our implementation of CSP, are described in the following sections.

5.1 Agent binding

As explained in Section 4.1.2 above, the `bind(`*args*`)` method of an `AgentBinder` object attempts to choose from an array of `AgentProcs` (the so-called *y* array) by means

of matching input arguments with the descriptors in the corresponding *x* array. Here we enlarge on the earlier sketchy explanation with the aid of a fuller example:

Suppose that there are three definitions of agent `P`: `P(1,1)`, `P(1,2)`, and `P(2,n)`. Now suppose there is an agent defined as follows:

```
Q ::= foo?bar(_i,_j)→P(_i,_j)
```

Which `P` definition to chain to is impossible for the translator to decide, so it codes a call to `P`'s `AgentBinder`, `P_b`. The two arrays generated along with `P_b` will look like this:

```
int P_x[] = {
   0,1, 1,1, AB_CALL,
   0,1, 1,2, AB_CALL,
   0,2, AB_CALL,
   AB_END };
AgentProc* P_y[] = { P_c1c1, P_c1c2, P_c2v };
```

`P_x`'s initialization has been broken out here by rows to show the purpose of each integer more clearly. The first row is a description of the arguments of the first element of `P_y`, that is, of `P_c1c1`, the `AgentProc` signature of `P(1,1)`. Within each pair of numbers, the first is an argument index (from 0) and the second is its required value. Thus the first row means "arg 0 is 1 and arg 1 is 1." The binder goes along the first row comparing the requirements with the actual values of input arguments `_i` and `_j`. If it reaches the symbol `AB_CALL` (an enumerated data type) without finding a contradiction, it chooses the first `AgentProc*` in `P_y`. But if an argument does not match, it skips past `AB_CALL` and tries the next row. If it manages to reach `AB_END`, it means that the input arguments did not match any of the descriptions, so a run-time error will be issued.

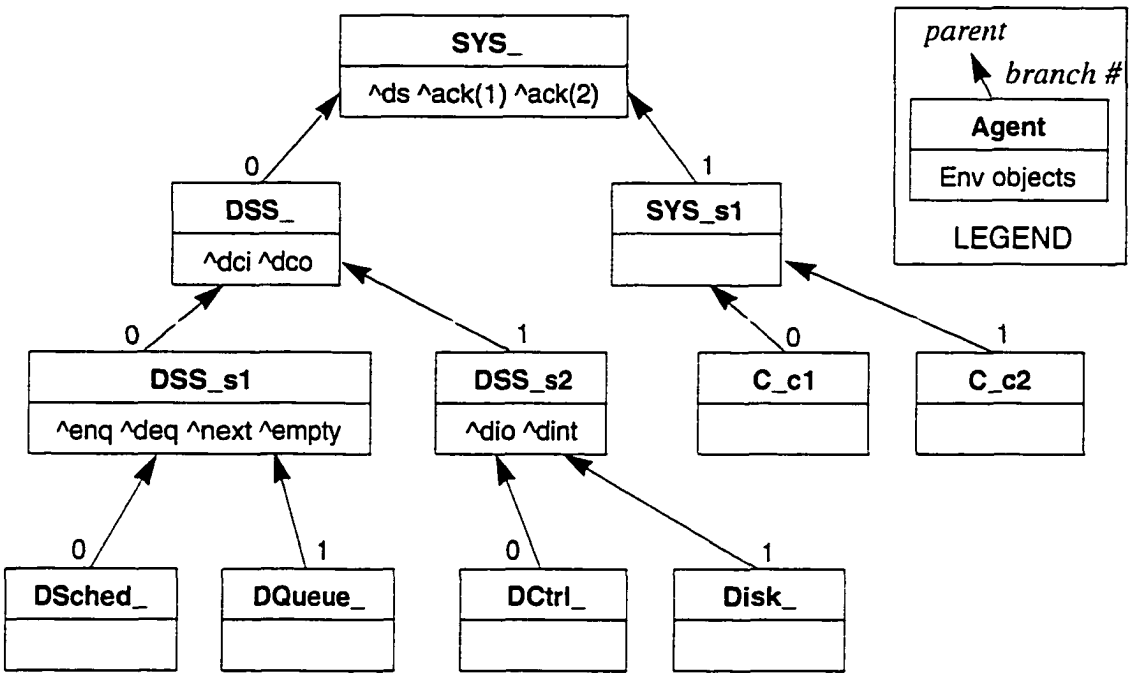
Some agents, like `P(2,n)` above, have a mixture of constant and variable arguments. In that case, their *x* descriptors do not mention the variable arguments, the effect being that *any* input value in the variable positions will match. Note that agent definitions must not have ambiguously overlapping arguments; that is, it would be an error to also define `P(n,2)`, since it would overlap both `P(1,2)` and `P(2,n)`.

5.2 Environment stack

What exactly happens when an action is executed by an agent? In CSP that depends, not only on the action's name, but on its *environment*. That environment is the accumulation of the concealment, renaming, and synchronization lists that have been piled up by the immediate agent as well as all its ancestors. Since agent descent (i.e., parents creating child tasks) in CSP is dynamic, not static, a run-time stack is needed to keep track of the environment.

The environment stack, also called the *agent descent tree*, adds a leaf node whenever a new `Agent` is created. Figure 8 is a snapshot of the tree after the first several agents of the disk server case study have been started by the two CSP statements printed below the tree.

Each box represents an `Agent` instance (including extracted subagents). Each `Agent` has its own section of the stack, containing whatever `Env` objects have been pushed while it is executing. In addition, each `Agent` object contains a pointer to its parent, so that stack searches can extend beyond the immediate `Agent` upward through all of its ancestors. The purpose of the *branch numbers* will be explained shortly.




```
SYS ::= ( DSS || (C(1) ||| C(2)) )^{ds,ack(1),ack(2)}
DSS ::= ( (DSched || DQueue)^{enq,deq,next,empty}
        || (DCtrl || Disk)^{dio,dint} )^{dci,dco}
```

Figure 8: Sample of agent descent tree

Overall this structure grows, or shrinks, like a tree, but since any given `Agent` only sees the nodes above it (toward the root `SYS`), the `Agent` considers that path to be its own personal environment stack. For example, an `Action` executing in the `DCtrl_Agent` (CSP statement not shown) would be subject to the total environment represented by the `EnvSync` objects within the shaded rectangles of Figure 8.

How the stack is involved with `Action` execution will be described in the next section.

5.3 Action execution

Figure 9 is a UML diagram of the classes mentioned in the rest of this chapter. It reveals more details than the class hierarchy in Figure 5 on page 44.

Whether an action is an `Atomic` or `Channel` type, it is executed by invoking the base class's `Action::execute()` method. The method carries out these steps of processing until the action has been "taken":

1. Check whether there is an *external routine* associated with the action. If so, call it and then return, considering the action "taken."
2. Otherwise, start searching the environment stack, from the current agent calling `execute()` upward through its parents. The actual searching is done by `Agent::searchEnv()`.

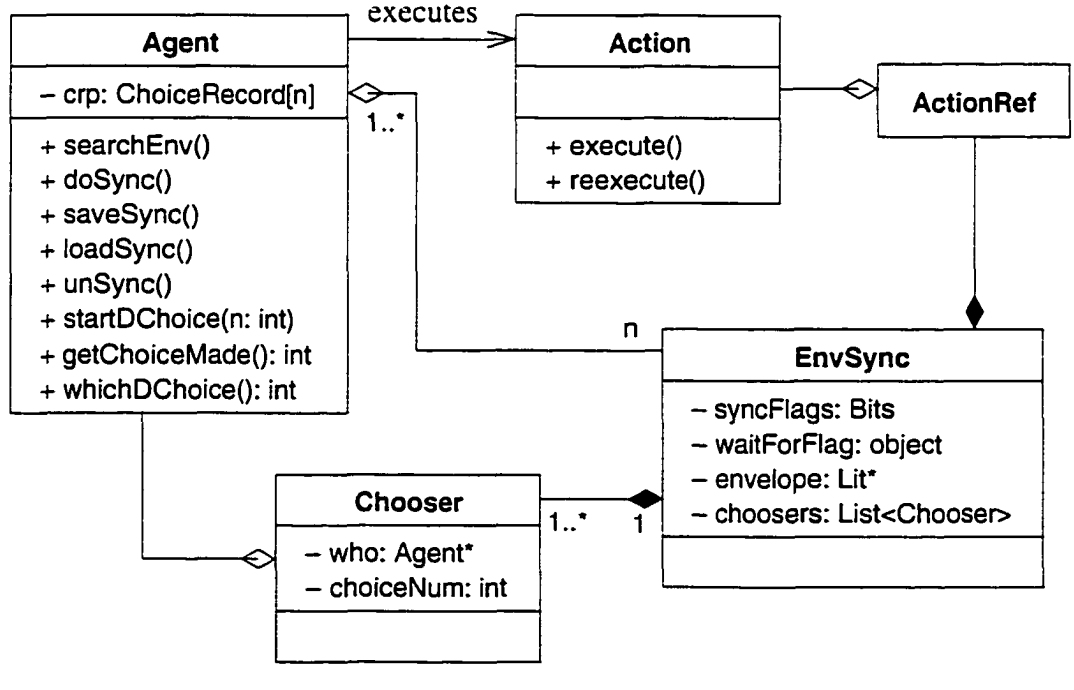


Figure 9: Details of classes involved in action execution

3. If an `Env` object is found having an `ActionRef` that matches this `Action`'s name,[1] process according to the `Env` subclass:

   - `EnvHide`: Consider the action "taken"; it can have no further effect.
   - `EnvRename`: Do the substitution and resume searching upward.
   - `EnvSync`: Call `Agent::doSync()` to attempt synchronization (see next section). After it succeeds, mark the action as "taken." If we are the active party to the syncchronization, resume searching.

The reason we resume searching after synchronization is in case a renaming appears farther up the stack. If so, this should affect the name of the trace. To avoid duplicate traces, our convention is that the "active" party (defined as the last agent to arrive at the sync, while the "passive" agents are sleeping) is responsible for printing the trace. The passive parties are silent.

When the stack search reaches its end, a trace will be printed showing the name of the current agent, its arguments, the name of the action, and its subscripts or channel data. If the tracing option is not selected on the command line (see Appendix B.4), then stack searching in support of tracing is bypassed in order to save execution time.

A more drastic aid to debugging is obtained by recompiling CSP++ with the `ACTWATCH` symbol defined (see Appendix B.1). This activates code that prints out the detailed steps of environment searching and matching on the `cerr` standard error I/O stream at run time.

---

1. This comparison, performed by `ActionRef::operator==()`, is not lexicographic, but simply compares `Action*` pointers, so it is very fast. When `Atomic ActionRefs` match, it is necessary to compare their subscripts too, so, for example, `ack(1)≠ack(2)`.

5.4 Multi-party synchronization

Each EnvSync object (Agent.h) on the environment stack is, in effect, a "nerve centre" for synchronization. It contains the following data members:

1. syncFlags: These flag bits (implemented by the Standard Components Bits class) are all 0 when no synchronization is in progress. When a synchronization attempt begins, one flag is reserved for each Agent that was composed below this level in the agent descent tree. Setting a flag to 1 signifies that an Agent is trying to synchronize on the Action represented by the EnvSync object.

For example, in Figure 8, the dint EnvSync object would have two flags, #0 for use by DCtrl_, and #1 for use by Disk_, or their respective descendants. If the Action dint occurs in Disk_, the usual environment search will discover dint in the stack of Agent DSS_s2. Since Disk_ is composed under that Agent's branch 1, Disk_ must use dint.syncFlags[1] when synchronizing.

2. waitForFlag: This single flag is an instance of object (task.h), so it can be waited for as described in Section 3.3.1 above. Agents that are not a party to an in-progress synchronization, but want to start a fresh synchronization as soon as the syncFlags are free, wait for this object to be alerted. This will happen when the current synchronization is finished, as part of the usual cleanup process.

The need for this flag is apparent from the following scenario, again referring to Figure 8. Suppose that client 1, Agent C_c1, has sent a message to the disk server via Channel ds, and is waiting for DSched_ to receive the message by synchronizing on this Channel. Now suppose that, meanwhile, client 2, C_c2, also wants to start some disk action. Both clients lie on the same agent descent tree branch below SYS_, so they

both must use `ds.syncFlags[1]` to communicate with `DSched_`. In this case, `C_c2` will find that "its flag" is busy, therefore it will wait its turn by having recourse to `ds.waitForFlag`.

3. `envelope`: This points to a heap `Lit`, to which, or from which, a `Channel` can copy data (required since the stack variables in the other agent may be out of scope when synchronization takes place).

The data structures in the agent descent tree have to be accessible to all the `Agent` tasks. In the LinuxThreads implementation, where preemption can occur at any time, they must be protected by a Pthreads mutex. The use of the global mutex `object::mlock`, shown in Figure 5 on page 44, was explained in Section 3.3.3. The convention is for the low-level synchronization method `Agent::doSync()` to lock this mutex before testing and manipulating flags, sleeping, or signaling other threads via `object::alert()`. This ensures the inviolability of the synchronization critical section in `doSync()`.

To summarize, an agent attempting to synchronize uses the above data structures in the following series of steps, carried out by `doSync()`. These are also depicted in the UML sequence diagram[2] [Pool99] of Figure 10, which illustrates a typical case of synchronization. In the figure, agents `Disk_` and `DSched_` are synchronizing on `Atomic dint`, and `Disk_` happens to arrive at the synchronization first. UML message sequence numbers are noted in brackets in the steps below.

---

2. We make one modification to the notation in [Pool99]. When an object directly accesses a data member of another class—in this case, an `Agent` accessing `EnvSync`'s members as a "friend class"—we draw the arrow that would normally indicate a message being sent to the accessed object, but we suppress the rectangle that indicates the activation of a member function.

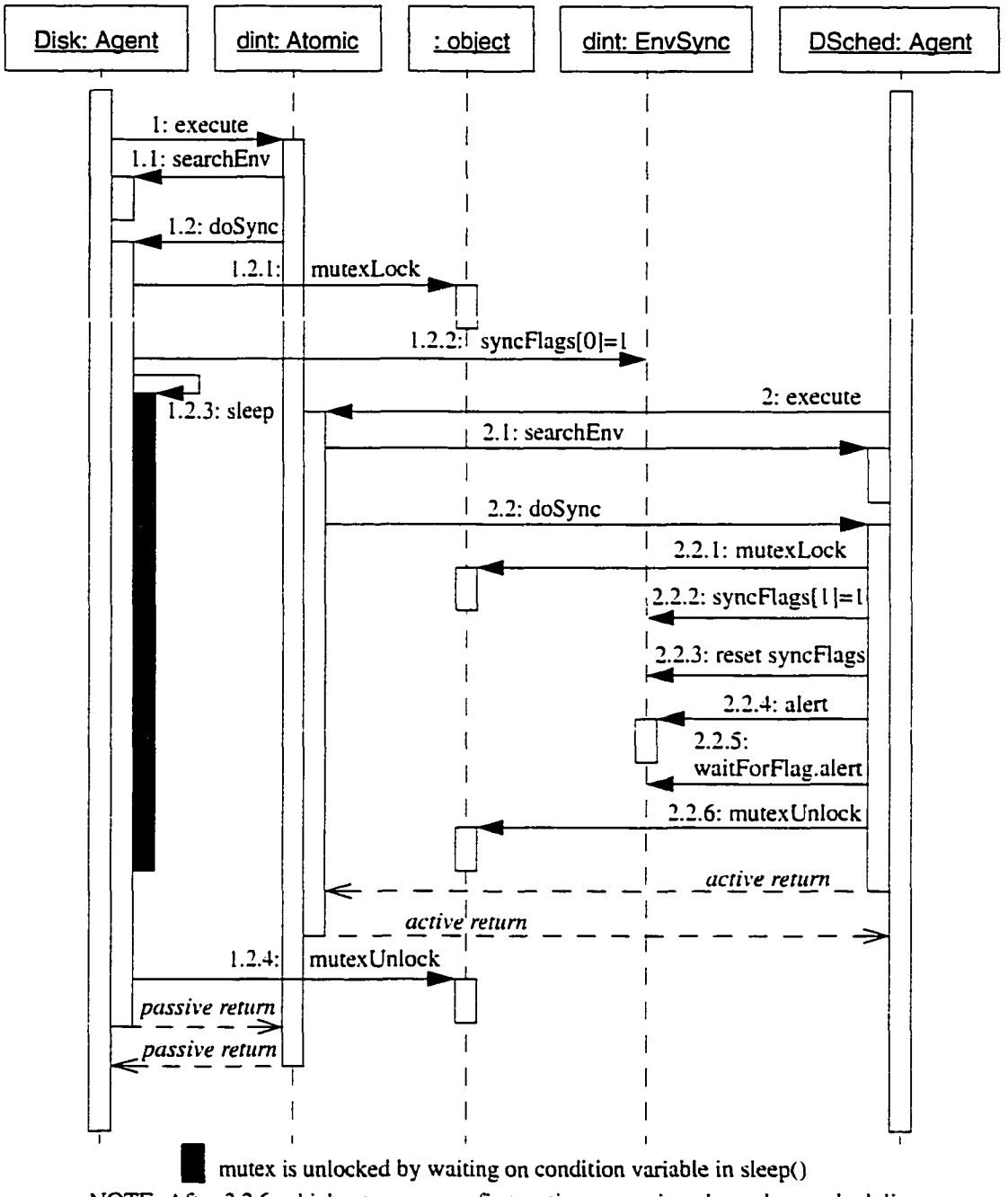


Figure 10: Sequence diagram of synchronization

1. *Try to get a syncFlag:* Since there may be other interleaved agents at this level, all trying to synchronize on the same action (named at a higher level), it is possible that the `syncFlag` reserved for this branch will already be in

use. If so, wait on `waitForFlag`. Upon wake up, repeat this test until the desired flag is found to be 0.

2. *Check in:* "Check in" for the synchronization by setting this branch's flag [1.2.2 and 2.2.2]. Then count the set flags to see whether synchronization has just been completed.

3. *Active return:* If so, we are the "active" party, responsible for cleanup and tracing. Clear the `syncFlags` [2.2.3] and wake up all `Agents` waiting either for a flag to become available (alert the `waitForFlag` object [2.2.5]) or for the synchronization to complete (alert the `EnvSync` object [2.2.4]).

4. *Passive return:* If there are other `Agents` still to check in, wait on the `EnvSync` object [1.2.3]. The last agent checking in will alert it [2.2.4].

5.5 Deterministic choice

Choice greatly complicates the straightforward picture painted above, where individual `Actions` were able to wait on their own for synchronization, thus suspending their `Agents` while they wait. But in the context of multiple alternative actions, if any one were to wait, the others then could not be tried. So in order to allow *all* actions to wait in concert, it is necessary to add a try-then-back-out apparatus to the synchronization mechanism.

When an `Agent` initiates an n-way deterministic choice by calling `Agent::startDChoice(`n`)`, an array of n `ChoiceRecord` objects (Agent.h) is created. These records will be needed for backing out and retrying. Each action is tried in turn, according to these steps:

1. If an earlier choice already succeeded (find out via `Agent::getChoiceMade()`), this one becomes a no-op; just return.

2. Otherwise, begin to carry out the steps of `Action` execution (Section 5.3) as usual. If the action is taken, then this choice succeeds (and others previously tried may need to be cancelled).

3. Otherwise, if synchronization is involved, `doSync()` is not allowed to wait as it normally would. Instead, it must take note of which *stage* the synchronization is at (i.e., waiting-for-flag or waiting-for-sync), and then call `saveSync()` to make a snapshot of this state in a `ChoiceRecord`. In addition, a `Chooser` object is created and attached to the `choosers` list of the `EnvSync` object to provide a back-pointer to this `Agent`. That pointer will later be used by the `Agent` completing the sync, in order to cleanup our choice state for us.[3]

When all choices have been tried, `Agent::whichDChoice()` is called. It determines which of these two situations holds:

1. *A choice succeeded:* The other failed `ChoiceRecords` are examined, and if any represents a synchronization that was started, it is cancelled by calling `Agent::unSync()`. Depending on the stage of the synchronization, `unSync()` will clear the branch's `syncFlag` (and alert any `Agents` waiting for it), or remove the `Agent` from the list of those waiting for access to the flag.

---

3. An essential point is that it cannot safely be left to agents to cleanup their own choice state. This is because scheduling, in the AT&T task library, is neither prioritized nor preemptible. When one sync action that is subject to an *n*-way choice completes, it is important to prevent any others from completing, by cancelling them at once. But there is no way to insure that a task wanting to cancel its own choices will get the CPU next, thus a race condition could result. This is prevented by having the synchronizing or "active" `Agent`, which already has the CPU, perform all cancellations on behalf of the passive `Agents`. That, in turn, requires back-pointers from the `EnvSync` object to all the choosing `Agents` that require the cancellation service.

In the LinuxThreads version, the possibility of preemption means that the cleanup has to be conducted with other threads locked out by means of the global mutex.

2. *No choice succeeded:* In that case, one or more of the tried actions must have initiated a synchronization. This `Agent` will sleep, waiting for a wake-up from the first synchronization that completes.

The scene now shifts to some other `Agent` task which is calling `Agent::doSync()` to complete one of the synchronizations we are waiting for:

1. When the sync is completed, the associated `choosers` list is examined to locate all parties who are involved in a choice.
2. For each `Agent` on the list, its `ChoiceRecords` are inspected in turn and `unSync()` is called for each one (except the choice that succeeded). All waiting agents are waked as usual.

Back now in `whichDChoice()` of the original choosing `Agent` task:

1. Reload the information in the `ChoiceRecord` which succeeded by calling `Agent::loadSync()`.
2. Call `Action::reexecute()` to continue processing the successful action that had been interrupted earlier. `reexecute()` will, in turn, call `doSync()` again.

Finally, at the end of all this bouncing back and forth, one synchronization will have been allowed to complete—it has been "chosen"—and all other synchronization attempts in the alternative choices will have been rolled back as if they had never been started.

Returning briefly to the theme of critical sections, it is obvious that in the context of trying several choices, the simple convention of `doSync()` capturing and releasing the mutex is insufficient when preemption can occur (LinuxThreads). The series of alternatives could then be interrupted by other `Agents`, and one which had just been determined *not* to have taken place might indeed have completed "behind the backs," so to

speak, of the subsequent choices prior to `whichDChoice()` being executed. The way to prevent this confusion is to modify the locking convention when a deterministic choice in effect: `startDChoice()` will capture the lock, and `whichDChoice()` will release it.

With the above description, the explanation of the CSP++ framework is essentially complete, though of course many details remain in the code. In the next chapter we return to our disk server case study, to show how it is actually translated and executed by the OOAF.

CHAPTER 6

Case Study and Experimental Results

The disk server case study was introduced in Section 2.1.2. Here we complete the case study by describing its translation and execution with CSP++. We then we present experimental results, obtained from a number of variations on the case study system that were evaluated in terms of resource consumption. For purposes of comparing our approach to "the legitimate competition," we reimplemented the disk server using a commercial CAD tool based on StateCharts. It will be seen below that CSP++ is actually competitive, despite being in its initial unoptimized version.

6.1 Disk server case study

In order to run a simulation of the disk server, we need to specify the behavior of some client processes for test purposes (refer to Figure 3 on page 15). For example:

```
C(1) = ds!req(1,100)→moreone→ack(1)→SKIP
C(2) = ds!req(2,150)→moretwo→ack(2)→SKIP
```

This has each client making a disk request, performing some more activity asynchronously (`moreone` and `moretwo`), waiting for acknowledgment from the server, and then terminating successfully (the special SKIP process in CSP).

Appendix A gives the complete C++ output of the translated disk server specification, including the two client processes shown above. This listing was produced using the "-s" source option (see Appendix B.2 user's manual) which conveniently copies the

csp12 source statements into the C++ file as comments directly before the generated agent definitions.

After the translated output is compiled and linked with the framework, we execute it with the "-t -i" trace and idle command line options (see Appendix B.4 user's manual). The trace option instructs the framework to print on `cerr` (stderr) the trace of every action taken. The idle option starts an additional agent that wakes up periodically and checks whether all the other agents are idle. If so, it aborts execution and dumps the framework's status to `cerr` (stderr).[1]

Running the DSSsim system produces the following output (line numbers added in brackets). Each line starting with "|=" is part of the trace. The name of the agent producing the trace (the one that completes the sync) is printed first, followed by the action taken. As in the traces output by the csp12 tool, "$" denotes a synchronization of channel communication, with the transferred data value following.

```
[1]   |=DS_idle [ds$req( 1, 100 )]
[2]   |=C( 1 ) [moreone]
[3]   Action: moreone
[4]   |=DCtrl [dci$start( 1, 100 )]
[5]   |=DS_busy [ds$req( 2, 150 )]
[6]   |=C( 2 ) [moretwo]
[7]   Action: moretwo
[8]   |=Disk [dio$100]
[9]   |=Disk [dint]
[10]  |=DQ( 0 ) [enq$req( 2, 150 )]
[11]  |=DS_busy [dco$fini( 1, 100 )]
[12]  |=DS_busy [ack( 1 )]
```

1. This idle-checking function was inherent in the initial AT&T coroutines version. In the Pthreads version, when the feature is wanted it has to be provided by an explicit thread.

```
[13]  |=DQ( 1 ) [deq]
[14]  |=DQ( 1 ) [next$req( 2, 150 )]
[15]  |=DS_check [dci$start( 2, 150 )]
[16]  |=DCtrl [dio$150]
[17]  |=Disk [dint]
[18]  |=DCtrl [dco$fini( 2, 150 )]
[19]  |=DS_busy [ack( 2 )]
[20]  |=DQ( 0 ) [deq]
[21]  |=DQ( 0 ) [empty]
[22]  idletask: All tasks IDLE!
[23]  Agent::exit_fn: Dump printed on stdout
[24]  == AGENT DUMP ==
[25]  Current # Literals: 1; High water marks:  9 Literals, 17 tasks
[26]  ============================================================
[27]  task #16 'DQ' (IDLE)
[28]  This task running as Agent DQ( 0 )
[29]   Waiting for sync on enq
[30]  My sync flag is #1 (LSB=#0) in [10]
[31]   Waiting for sync on deq
[32]  My sync flag is #1 (LSB=#0) in [10]
```

From this we can observe how client `C(1)` submits to the disk scheduler a request for disk block 100 (line 1) and then continues about its business (line 2). "`Action: moreone`" (line 3) is the default output from an action that has no external routine provided, but neither is trapped in the environment for synchronization. These default actions are printed on `cout` (stdout), independent of whether tracing is enabled, and are useful as stubs for external routines until they can be written and linked in.

`C(1)`'s request is passed to the disk controller (line 4). We observe how the disk scheduler goes from the IDLE state (line 1) to BUSY (line 5), and receives `C(2)`'s request for block 150. The second request is queued, but the trace for this action doesn't appear until line 10. The action is taken by agent `DQ(0)`, signifying that there were pre-

viously zero requests in the queue.

Meanwhile, C(2) continues (lines 6-7). At this point, the disk controller orders the disk drive (via channel dio) to access block 100 (line 8) to carry out the first request. After some time, the disk interrupts (action dint), the controller notifies the scheduler, and the scheduler acknowledges client C(1) (lines 9, 11, and 12). We then observe them repeat the cycle with the second request (lines 13-19), which was enqueued while the server was busy. (The inner queue actions are not traced because the CSP code ordered them hidden.)

Finally, things come to a halt when the disk scheduler checks the queue again and finds it empty (lines 20-21). Since there is no ready-to-run task, the idle-check agent activated by the "-i" option calls the exit function, which prints a dump of all the existing agents (only the first one is reproduced here). This same dump can be produced by an agent transferring to the special agent STOP. Useful information includes maximal resource usage ("high-water marks") and precise agent status. In the case of the task shown above, we can see that its identity is agent DQ(0), and it is waiting for synchronization to occur on either of two actions, enq or deq. The sync flags from those two EnvSync objects are displayed, and we observe the agent's flag (the left-hand bit) set, just as it should be, since DQ(0) has arrived first at the syncs.

Following the above example in Appendix A, there is a second worked example which suggests how system development can proceed. In the second system, the simulated Disk agent has been removed from the specification of agent DSS. Now, the actions that previously synchronized with Disk, i.e., the channel dio and the atomic action dint, go out to the environment. If no external routines were provided to link

with them, default actions would be logged, just as with actions `moreone` and `moretwo` in the trace above. However, in this example C++ procedures are provided. They are linked in by means of preprocessor symbols during compilation (see Appendix B.3 for exact procedure). When the DSS system is executed, the routines are called whenever the `dio` and `dint` actions occur, as shown in the trace log (Appendix A.6).

6.2 Commercial CAD tool comparison

In order to put CSP++ into perspective as a tool for code generation, it is helpful to compare it with something similar. One is tempted to produce a hand-coded C++ program, say of the disk server case study, and let it go head-to-head with the program generated by CSP++. The main theoretical objection to this kind of comparison is that there is no straightforward way for a programmer to proceed, starting from either the CSP or StateCharts (Figure 3 on page 15) specification of the system. Basically, one has two choices when faced with a non-trivial specification:

1. One can preserve the specification's computational model—that is, concurrent synchronizing processes, or concurrent hierarchical finite state machines—in which case one is forced to construct the infrastructure required to support the model's execution.
2. Alternatively, one can use the specification as a behavioural model for designing a program using a different computational model. This would be one whose infrastructure is inherent in the chosen programming language, i.e., a conventional structured design or an object-oriented design.

As for the first choice, this is exactly what has been accomplished in creating the CSP++ OOAF for the general case. There is little point in carrying out the exercise again

by hand for a specific case. The problem with the second approach is that without a methodology for converting an analysis in one computational model into a design in another model, the result is apt to be extremely arbitrary and may not be significantly representative of "hand-coded programs." Again, it is the exact purpose and achievement of this research to formulate a methodology for converting CSP specifications to C++ programs, and to do it automatically. To find another way of doing the same thing manually, does not really qualify as "comparison to hand-coded C++."

Leaving aside comparisons that have superficial appeal but questionable value, we turn to a more relevant arena, that of CAD tools which have a similar purpose to CSP++. ObjecTime Limited produces a commercial tool expressly for C or C++ code generation from StateChart models, or more precisely, the adaptation known as ROOMcharts [Seli93]. ObjecTime Developer claims to be "the only object-oriented software development toolset designed specifically for event-driven, real-time systems" [OTI]. Implementing the disk server case study in ObjecTime is a meaningful comparison, first, because we do not have to change the computational model (we already have a StateChart diagram in Figure 3, and StateCharts are fairly compatible with CSP), and second, because the output of the tool is also executable C++ code. We can even do the timing on the same hardware, albeit under different operating system environments.

6.3 Timing tests

Using the disk server case study as a baseline, repetitions were introduced to inflate its execution time to a significantly measurable level. These test cases are laid out in Table 2, along with the average execution time obtained on a 400 MHz Pentium II with 128Mb of

memory, running Red Hat Linux 6.2. The g++ compiler used was ecgs-2.91.66, with -O2

Test Case Description	User Secs.	System Secs.	Total Secs.

(1) 20,000 disk accesses in 20,000 process creations
 	
`C(1) ::= ds!req(1,100)->ack(1)->SKIP.`
`C(2) ::= ds!req(2,150)->ack(2)->SKIP.`
`Test(_i) ::=`
`(if _i>0 then ((C(1)|||C(2)); Test(_i-1)))`
`+ STOP.`
`SYS ::= (DSS||Test(10000))^{ds,ack(1),ack(2)}.`
	6.33	4.45	10.78

(2) 20,000 disk accesses, synchronized in pairs, in 2 process creations
 	
`C(1,_n) ::=`
`(if _n>0 then ds!req(1,100)->ack(1)->syncC`
`->C(1,_n-1)) + SKIP.`
`C(2,_n) ::=`
`(if _n>0 then ds!req(2,150)->ack(2)->syncC`
`->C(2,_n-1)) + SKIP.`
`Test(_i) ::= (C(1,_i)||C(2,_i))^{syncC}; STOP.`
`SYS ::= (DSS||Test(10000) )^{ds,ack(1),ack(2)}.`
	1.60	1.25	2.85

(3) 20,000 disk accesses; same as (2) but syncC removed from clients `Test(_i) ::= (C(1,_i)|||C(2,_i)); STOP.`	1.65	1.24	2.89
(4) 10,000 disk accesses; same as (1) with `Test(5000)`	3.20	2.16	5.36

Table 2: Timing test results

optimization. Timing was obtained by running the executable with the "-q" option (to prevent a status dump; see Appendix B.4) under control of the tcsh (C shell) **time** command. Each test was run five times, and the timings averaged.

6.3.1 Test results

The first case introduces a `Test` process into the system specification to drive the repetitions. The rest of the code mirrors the baseline, except that the `moreone` and `moretwo` actions, which just result in printing on the console, have been removed. Since Linux

deals with POSIX threads directly in the kernel, there is a substantial system time component in the total execution time: 41%. In contrast, when this same test was run on a Solaris system, where the POSIX threads are mostly managed in user space, system time drops to under 1%.

It was observed that in test (1), 20,000 processes are being created and destroyed as the `Test` process loops composing the clients in `C(1)|||C(2)`. In test (2), the 10,000-cycle loop is moved from the Test process down into the clients themselves. A new `syncC` action is introduced in order to synchronize the disk requests in pairs, to avoid overflowing the primitive disk request queue. Now, `C(1)` and `C(2)` are created only once each. One would expect the thread management overhead to decrease accordingly, and indeed the results show a dramatic drop in execution time. Interestingly, the proportion of system time vs. total is about the same as before (44%).

The purpose of test (3) was to see whether pairwise synchronization was really required. It was not, though removing the extra `syncC` action has hardly any appreciable effect on the timing. Tests (2) and (3) seem to be the "best case" timing that can be obtained for 10,000 repetitions (20,000 simulated disk accesses) by simple process restructuring.

Test (4) cuts the repetitions of test (1) in half to see whether the time for looping is scaling linearly, as one would hope. The results, almost exactly one-half of (1), shows that this is the case.

An attempt was made to make the simulation more realistic by incorporating a new `delay(`*msec*`)` action into the `Disk` process, implemented by an external procedure call-

ing the Linux nanosleep() function. This did produce the desired delay, and allowed for rescheduling between requests, which made for more consistent exercise of the disk request queue processes (BUFF and CELL). Another test was created where BUFF and CELL were replaced by an efficient user-coded queue. Unfortunately, calling nanosleep() or sleep() seemed to reset the Linux timekeeping to zero, and it was impossible to obtain any useful measurements this way.

6.3.2 Comparison with ObjecTime

The model built using ObjecTime Developer 5.2.1 mirrored the StateCharts diagram (Figure 3) as far as possible, with additional behaviour filled in from the CSP specifications. The printout of the structural and behavioural models is given in Appendix E. The tool was used to generate C++ code to run under control of the ObjecTime real-time executive (Micro Run Time System Release 5.21.C.00). It was compiled using Microsoft Visual C++ 6, and executed in a DOS window under NT4. The hardware platform was identical to that used for the CSP++ time trials under Linux.

In order to set up a test case comparable to those above, the test harness behaviour in the outermost block triggered the clients 10,000 times, thus resulting in 20,000 disk requests, as in the CSP++ version. Timing data was obtained by calling the ANSI C clock() function to return the elapsed CPU time at the start and end of model execution. The difference of start and end times of five runs was averaged to get the result of 3.76 CPU seconds.

6.3.3 Analysis

The key inference coming from the timing data concerns the overhead inherent in the CSP++ OOAF as it is currently implemented. The helpful breakout of user versus system times in Linux shows clearly that the framework's overhead—consisting of thread creation, thread scheduling, mutex locking and unlocking, and condition variable waiting and signalling—is at least 40%. This is therefore a ripe area to target for optimization.

The purpose of the ObjecTime comparison is to show whether the CSP++ execution times are *reasonable* in light of state-of-the-art code generation tools. As a matter of fact, they compare quite favourably with the ObjecTime results. ObjecTime ran faster than test (1), but when the huge amount of gratuitous process creation was cut out in tests (2) and (3), the CSP++ program finished first. Considering that ObjecTime is an expensive commercial tool (licensed at over US$15,000 per seat), and the company has had years to optimize its real-time executive, the result obtained by the initial version of CSP++ running under generic desktop PC Linux is quite gratifying.

It is also worth noting that the graphical entry method of ObjecTime model construction was found to be exceedingly slow and tedious compared with the simple textual entry method of CSP++. This is similar to the contrast between schematic capture versus hardware description language methods for entering circuits in modern CAD tools, and helps explain why in recent years the graphical methods have been largely overshadowed by the textual for large designs.

6.4 Memory estimates

Sizes for the object files that make up test (1) are listed in Table 3, as obtained via the GNU size utility. This does not tell the whole story, since dynamically linked modules are

	Code Sections			
Filename / category	**Text**	**Data**	**Bss**	**Total**
Framework files				**15988**
Action.o	4175	148	0	4323
Agent.o	5062	116	0	5178
Lit.o	2842	92	0	2934
task.o	3317	232	4	3553
SC-3.0 library				**5805**
Bits.o	2644	0	4	2648
List.o	2735	0	0	2735
Pool.o	422	0	0	422
Translated test (1)				
DSSsim.o	8874	320	504	**9698**
Runtime library				**4814**
Executable				
DSSsim	34145	1036	1124	**36305**

Table 3: CSP++ object file sizes

pulled in from the C++ and POSIX threads shared object libraries at execution time. And of course, the heap for dynamically allocated variables and per-thread stacks are not accounted for. Still, the basic code size of both the framework and the translated application are relatively modest. The approximately 21K of code and data is the fixed framework component (including the SC-3.0 library routines) that will not vary from application to application.

At this stage of development, no attempt has been made to control the size of the heap and the stacks. The system defaults have been allowed to take their course. How-

ever, there is an option for creating POSIX threads with a user-specified stack size, and since there is no recursion in the OOAF, it should be possible to determine a safe maximum. The dump provided at the end of a CSP++ execution states the "high water mark" for tasks (i.e., threads) and dynamically allocated literals during the course of execution. In the case of test (1) these numbers are 14 literals and 16 tasks. Multiplying the latter by the maximum stack size would yield the total memory requirement for stack space. Similarly, we can put an upper bound on heap requirements through knowing the maximum number of literal and task objects to be allocated.

Turning back to ObjecTime, this unfortunately degenerates into an "apples and oranges" kind of comparison with regard to memory use. A crude comparison can be made of the executables. The DiskServer Sys.exe file includes 131K of code and 37K of data, for a total initial footprint of approximately 168K. Superficially this appears to be nearly five times the size of the analogous CSP++ DSSsim executable. On the other hand, Sys.exe executes from a DOS prompt, and already contains the linked run-time library modules. Furthermore, the real-time executive includes a primitive debugger interface, which is taking up space. The ObjecTime documentation gives complex instructions for estimating the runtime memory requirements of a given application, but the calculations are difficult to carry out. More detailed analysis would be required to make meaningful comparisons in this area.

In summary, the studies above show that CSP++ is on the way to being an efficient software synthesis tool. There is certainly room for optimization, both in execution time and memory use, and these can be carried out with later versions under Future Work.

CHAPTER 7

Conclusions and Future Work

In light of the work described in the preceding chapters, the two main questions which can now be answered are these:

1. *Did we succeed in making CSP specifications executable, in the software synthesis sense?*
2. *Is there value in the OOAF approach to software synthesis?*

These questions are discussed below. In addition, a number of avenues of future work will be put forward. Finally, we report the exact status of our work to date, for the sake of those who wish to explore, utilize, or expand on it, including the availability of CSP++ to the public.

7.1 Conclusions

7.1.1 Proof of concept demonstrated

The answer to the first question above is "yes": By means of the cspt translator and run-time framework, we showed that CSP specifications can be executed as a C++ program and their traces printed, just as if being simulated by a verification tool. Most important, we are able to fully support the essential features of deterministic choice with multi-party synchronization. Furthermore, we showed how to link CSP specifications with user-coded procedures by identifying the latter with CSP actions. This enables CSP specifications to be directly transformed into a C++ control program for a larger software system.

7.1.2 Viability of OOAF approach

We consider that the OOAF approach has also been shown to be valuable. Its strengths lie in two particular areas:

1. *The framework's elements provide a good high-level code generation target, much easier to translate to, compared with conventional object code or assembly language targets, or even a procedural high-level language (HLL).*

Translating from the source language—CSP, in this case—to C++ invocations of the framework is a relatively short step both syntactically and semantically, and allows the burden of code generation to be largely shifted onto a existing HLL compiler (here, g++).

This approach is faster, simpler, and more maintainable than writing a conventional compiler: faster and simpler, because the job has been subdivided into two more tractable pieces; more maintainable, because the run-time system is itself HLL, not assembly language, and it is collected in one place, not dispersed throughout the translated target output. The approach is analogous to that taken with Java: writing a Java compiler that produces byte code, and writing a Java Virtual Machine (the run-time environment) that executes it. This is "faster, simpler, and more maintainable" than writing a number of Java native code compilers.

2. *The OOAF has high portability.*

This characteristic was illustrated by the conversion of the task model—arguably the most sensitive aspect of the run-time system—with practically no disturbance of the framework's source code, and no change to the translation algorithms.

That OOAF technology should prove valuable in the specialized context of soft-

ware synthesis is interesting, because it is not a "traditional" use (inasmuch as anything in so new a field can be called that) of the technology.

7.2 Future work

There are a number of potentially fruitful directions in which this research can be carried forward. These fall roughly into three categories, listed in order of increasing scope:

1. Consolidating the current work, that is, maturing it from "proof of concept" stage to the status of a robust and well-exercised tool more likely to be used for something beyond academic experimentation.
2. Extending the power of CSP++ by incorporating capabilities of other dialects of CSP.
3. Applying the OOAF technique to other formalisms.

One area which is *not* recommended, although it may readily spring to mind, is the job of finishing up support for the partially implemented and unimplemented constructs of csp12. The rationale for this advice is explained in the first section below.

7.2.1 Integration with commercial model-checker

While csp12-style CSP has been an excellent baseline for proof-of-concept development, not to mention entailing zero cost, the limited prospects for ongoing support of this in-house tool make it a less strategic underpinning for future work. Instead, a commercially-supported tool should be procured and time should be invested in realigning the front end of cspt to accept its syntax. FDR is the obvious choice. Taking this as the first step in any future work will obviate the need to deal with the unimplemented and partially imple-

mented csp12 constructs (Sections 4.4 and 4.5) that do not occur in FDR's dialect. This recommendation falls squarely under plans for "consolidation."

7.2.2 Enhancement of user code interface

The design of the framework's interface to user-coded action procedures is relatively immature, and ought to be addressed in order to consolidate the work. Now that a proof-of-concept has been achieved, what is lacking is a more realistic case study, that would also include simulation and model checking. An good example of this kind of effort, which is no small undertaking, is an RS-232 character repeater design by the Naval Research Laboratory [Moor96]. Moore and Payne used CSP for the design, and two tools for model checking, FDR and EVES.[1] One could even start by duplicating their design and synthesizing it with CSP++. Chapter 7 of [Hinc95] also contains a lengthy case study based on a network communications protocol.

The main point is that in the course of working through such a case study, it will become apparent in what ways the action procedure interface needs to be enhanced. For example, the issue of handling interrupts must be considered. Of particular interest will be the precise means of making user-coded procedures participate in deterministic choices. These changes will likely involve the OOAF itself, but not the translator.

7.2.3 Adaptation to other CSP dialects

It was mentioned earlier that CSP++, and indeed CSP itself, have no way of dealing with timing constraints. This can be remedied by implementing constructs from Timed CSP.

---

1. EVES, a formal methods tool not specific to CSP, is available via free download from ORA Canada [EVE].

Sections 4.1ff. of [Hinc95] have a good discussion of these operators and their algebraic properties. The key addition is a new built-in "`WAIT` *n*" process that delays execution for *n* time units. A new *timeout* operator is also defined: $P \overset{t}{\triangleright} Q$ runs as process P for no more than *t* time units, whereupon it switches to process Q.

The TOSCA tool [Balb96] shows another way to introduce timing into a CSP-like environment (in their case, Occam II): Simply provide a special channel that outputs an integer representing the current time. This gives agents, in effect, a way to read the system's real-time clock just by executing an action, e.g., `clock?_time`.

In a related area, some control could be given to agents over their scheduling priority. Currently, all agents are created as equal-priority threads. POSIX threads provide for dynamic priority adjustment, and this feature could be made accessible to agents by means a special process or action. This could meet the needs of some real-time systems.

Some of these extensions will require modifications to both the OOAF and the translator.

7.2.4 Optimization of resource usage

For highly-resource constrained applications, CSP++ would be more attractive if its resource usage could be reduced. There are many avenues for optimization that can be explored, such as the following:

- Discontinue the use of C++ I/O classes to communicate with the user, as this pulls in numerous bulky modules from the library.
- Port the framework to an economical real-time kernel that supports C++. Alternatively, write a simple scheduler along the lines of the AT&T coroutine

library. Returning to a coroutine threading model has the potential to markedly reduce system overhead (locking, waiting, preempting, etc.).

- Calculate actual stack requirements, instead of relying on OS defaults.
- Perform static analysis on the process structure, so as to determine action binding for synchronization purposes at translation time. In cases where this succeeds, run-time searching of the environment stack could be eliminated.

No doubt many other ways to reduce memory and CPU usage will be found when optimization is seriously attempted.

7.2.5 Adaptation to other formalisms

Finally, it would be interesting to apply the OOAF approach underlying CSP++ to other formalisms. CCS is a good place to start, both because of its similarity to CSP and the availability of commercial verification tools comparable to FDR. For formalisms that have CSP-like semantics, it may be possible to rework the lexical/syntax phase of the translator to build, say, a CCS parse tree, and then convert it to an analogous tree built with csp12 parse nodes. The back end of the existing translator would then generate C++ code based on the CSP++ framework classes, and thereby execute the CCS specifications. Alternatively, a CCS-specific framework could be created, using CSP++ as a source of design patterns. The latter approach would be suitable for formalisms that are semantically remote from CSP.

7.3 Status and availability of CSP++

The details in this dissertation are based on the latest version of CSP++ numbered V2.1, which is available to the public by anonymous FTP from the author's web site [Gard]. It

will compile and execute on Linux and Solaris, and likely on any platform having both a current C++ compiler and OS supporting POSIX threads. Table 4 gives the complete picture of the various versions of CSP++ and where they can be obtained. Source code for some variation on the Disk Server case study is included in each distribution.

Task Model	Version	Platforms	Contents	Source Code	Documentation
USL task library	1.0	Sun/SunOS, AT&T cfront	OOAF	CD ROM accompanying [Faya99]	[Gard99a] Tech Report (PDF file on CD)
POSIX threads	2.0	PC/Red Hat Linux, g++	OOAF		
	2.1	PC/Red Hat Linux & SPARC/Solaris, g++	OOAF & cspt	FTP from web site [Gard]	Dissertation (PDF file)

Table 4: Availability of CSP++ software

Bibliography

[Alag98] V.S. Alagar and K. Periyasamy. *Specification of Software Systems*. Springer-Verlag, 1998.

[Arro94] B. Arrowsmith and B. McMillin. How to program in CCSP. Technical Report CSC 94-20, Department of Computer Science, University of Missouri-Rolla, August 1994.

[Aust99] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison Wesley Longman, 1999.

[Balb96] A. Balboni, W. Fornaciari, and D. Sciuto. Partitioning and exploration strategies in the TOSCA co-design flow. In *Proc. Fourth International Workshop on Hardware/Software Codesign*, pages 62–69, Pittsburgh, March 1996.

[Berg85] J.A. Bergstra and J.W. Klop. Algebra of Communicating Processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[Budd97] Timothy Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, second edition, 1997.

[Carr96] C. Carreras, J.C. López, M.L. López, C. Delgado-Kloos, N. Martínez, and L. Sánchez. A co-design methodology based on formal specification and high-level estimation. In *Proc. Fourth International Workshop on Hardware/Software Codesign*, pages 28–35, Pittsburgh, March 1996.

[Chen94] Mantis H.M. Cheng. Communicating Sequential Processes: a synopsis. Dept. of Computer Science, Univ. of Victoria, April 1994.

[CSP] CSP archive, Programming Research Group, Oxford University. http://www.comlab.ox.ac.uk/archive/csp.html [as of 5/29/00].

[Davi92] J. Davies and S. Schneider. A brief history of timed CSP. Technical Report PRG-96, Programming Research Group, Oxford University, April 1992.

[Donn92] Charles Donnelly and Richard Stallman. *Bison: The YACC-compatible Parser Generator.* Free Software Foundation, Cambridge, MA, December 1992. Version 1.20.

[EVE] EVES web site, ORA Canada. http://www.ora.on.ca/eves.html [as of 5/29/00].

[Faya97] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[Faya99] M. Fayad, D. Schmidt, and R. Johnson, editors. *Implementing Application Frameworks: Object-Oriented Frameworks at Work.* John Wiley & Sons, 1999.

[FDR] FDR web site, Formal Systems (Europe) Limited. http://www.formal.demon.co.uk/FDR2.html [as of 5/29/00].

[FSE] Formal Systems (Europe) Limited, corporate web site. http://www.formal.demon.co.uk [as of 6/23/00].

[Gajs94] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems.* P T R Prentice Hall, 1994.

[Gamm95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gard] William Gardner, home page. http://faith.csc.twu.ca/~wgardner [as of 5/29/00].

[Gard99a] William B. Gardner. Synthesis of C++ software from CSP specifications. Technical Report DCS-252-IR, Rev. A, Dept. of Computer Science, Univ. of Victoria, June 1999.

[Gard99b] William B. Gardner and Micaela Serra. *CSP++: A Framework for Executable Specifications*, chapter 9. In Fayad et al. [Faya99], 1999.

[Hild97] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In *Proc. of the 20th World Occam and Transputer User Group Technical Meeting*, pages 48–76, Enschede, The Netherlands, 1997. http://www.rt.el.utwente.nl/javapp/cjt/CJT-paper.PDF [as of 6/23/00].

[Hinc95] Michael G. Hinchey and Stephen A. Jarvis. *Concurrent Systems: Formal Development in CSP.* McGraw-Hill Book Company, 1995.

[Hoar85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Lea96] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 1996.

[Lewi95] Ted Lewis, editor. *Object-Oriented Application Frameworks.* Manning Publications Co., Greenwich, CT, 1995.

[Lewi98] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads.* Sun Microsystems Press, Prentice Hall, 1998.

[Logr92] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992.

[Luqi97] Luqi and Joseph A. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, January 1997.

[LXT] LinuxThreads web site. http://pauillac.inria.fr/~xleroy/linuxthreads/ [as of 5/29/00].

[Mats93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[McGh98] Harlan McGhan and Mike O'Connor. Picojava: A direct execution engine for Java bytecode. *Computer*, 31(10):22–30, October 1998.

[Miln95] R. Milner. *Communication and Concurrency.* Prentice Hall, 1995.

[Moor96] Andrew P. Moore and Charles N. Payne. The RS-232 character repeater refinement and assurance argument. NRL Memorandum Report 5540-96-7872, Naval Research Laboratory, Washington, DC, July 1996. http://www.itd.nrl.navy.mil/ITD/5540/publications/CHACS/1996/newrptrTL/node1.html [as of 6/23/00].

[OTI] ObjecTime Limited, corporate web site. http://www.objectime.com [as of 6/23/00].

[Paxs95] Vern Paxson. *Flex, version 2.5: A fast scanner generator.* University of California, Berkeley, 2.5 edition, March 1995.

[Pool99] Rob Pooley and Perdita Stevens. *Using UML: Software Engineering with Objects and Components.* Addison Wesley Longman, 1999.

[Pres97] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, fourth edition, 1997.

[Reit97] Stephan Reitzner. Splitting synchronization from algorithmic behaviour. Technical Report TR-I4-97-08, Friedrich-Alexander-University, Erlangen-Nürnberg, Germany, April 1997.

[Rosc94] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice Hall International Series in Computer Science, pages 353–378. Prentice Hall, 1994.

[Rosc98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[Scat95] Bryan Scattergood. Book review: Concurrent Systems: Formal Development in CSP by Hinchey. *C Vu*, 7(6), September 1995. The Association of C and C++ Users. http://www.accu.org [as of 6/23/00].

[Seli93] Bran Selic. An efficient object-oriented variation of the Statecharts formalism for distributed real-time systems. In *CHDL '93: IFIP Conference on Hardware Description Languages and Their Applications*, Ottawa, April 1993.

[Silb98] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts.* Addison-Wesley, fifth edition, 1998.

[Somm96] Ian Sommerville. *Software Engineering.* Addison-Wesley, fifth edition, 1996.

[Srin99] Savitha Srinivasan. Design patterns in object-oriented frameworks. *Computer*, 32(2):24–32, February 1999.

APPENDIX A

Source Code for Disk Server Case Study

The following listings are provided in this appendix:

1. DSSsim example, with simulated disk:
   - Input to cspt (file DSSsim.csp12): csp12 specification (Section A.1 on page 122)
   - Parse tree produced by "cspt -t" translation (Section A.2 on page 123)
   - C++ output (file DSSsim.cc) produced by "cspt -s" translation (Section A.3 on page 127)

It will be helpful to view these listings in conjunction with the Statecharts depiction of the Disk Server, found in Figure 3 on page 15, and the run-time trace given in Section 6.1 on page 98.

2. DSS example, with external procedures in place of the simulated disk:
   - Input to cspt (file DSS.csp12): csp12 statements showing `Disk` agent removed from `DSS` specification (Section A.4 on page 134)
   - C++ source code for external routines (file DiskProcs.cc) (Section A.5 on page 135)
   - Run-time trace showing invocation of external routines (Section A.6 on page 135)

A.1 Csp12 specification (DSSsim)

```
% This is the simulated Disk Server, with simulated Disk and
simulated clients.


%============================
% DQueue:  disk request queue
%
% Interface:
%   enq!<item>enqueue item
%   deq     dequeue item, followed by:
%     next?_x  next item returned, or
%     empty  empty queue indication
%============================

CELL ::= left?_x -> shift -> right!_x ->CELL.

% BUFF ::= CELL |> CELL |> CELL .just 2 cells for now
BUFF ::= (((CELL#{right=comm}) ||
(CELL#{left=comm}))^{comm})\{comm}.

DQueue ::= ((DQ(0) || BUFF)^{left, right, shift})\{left,
right, shift}.

DQ(_i) ::= enq?_x -> ( left!_x -> shift-> DQ(_i+1) )
       | deq -> ( (if _i=0 then empty -> DQ(0))
            + fix X.(right?_y -> ( next!_y -> DQ(_i-1) )
                | shift -> X)
            ).

%============================
% DCtrl:  disk controller
%
% Interface:
%   dci!start(_cl, _blk)start operation on block <_blk> for
client <_cl>
%   dco?fini(_cl, _blk)operation finished
%============================

DCtrl ::= dci?start(_i, _blk)-> dio!_blk-> dint ->
dco!fini(_i, _blk) -> DCtrl.

%============================
% Disk:  disk drive (simulated)
%
% Interface:
%   dio!_blkperform disk i/o on block _blk
%   dint       disk interrupt signalled
%============================

Disk ::=  dio?_blk -> dint ->Disk.

%============================
% DSched:  disk scheduler
%
% Interface:
%   ds!req(_cl, _blk)client <_cl> requests operation on block
<_blk>
%   ack(_cl)client's operation finished
%============================

DSched ::= DS_idle.

DS_idle ::= ds?req(_cl, _blk) -> dci!start(_cl, _blk) ->
DS_busy.

DS_busy ::= dco?fini(_cl, _blk) -> ( ack(_cl) -> DS_check )
        | ds?req(_cl, _blk) -> enq!req(_cl, _blk) -> DS_busy.

DS_check ::= deq -> ( empty -> DS_idle
```

```
          | next?req(_cl, _blk) -> dci!start(_cl, _blk) ->
DS_busy ).
%============================
% DSS:  disk server subsystem
%
% Interface: (see DSched)
%============================

DSS ::= (  (DSched || DQueue)^(enq,deq,next,empty)
    ||
      (DCtrl || Disk)^(dio,dint)   )^(dci,dco).

SYS ::= (DSS || (C(1)|||C(2)) )^(ds,ack(1),ack(2)).

%============================
% Demo
%============================

C(1) ::= ds!req(1,100)->moreone->ack(1)->SKIP.

C(2) ::= ds!req(2,150)->moretwo->ack(2)->SKIP.
```

A.2 Syntax tree (DSSsim)

```
Reading from file: DSSsim.csp12
Translating to file: DSSsim.cc
Got definition: CELL ::= ( prefix
   ( input
      Channel left
      _x
   )
   ( prefix
      shift
      ( prefix
         ( output
            Channel right
            _x
         )
         Agent CELL
      )
   )
)
Got definition: BUFF ::= ( Env H
   ( Env S
      ( compose
         ( Env R
            Agent CELL
            ( rename from ... to
               Action right
               Action comm
            )
         )
         ( Env R
            Agent CELL
            ( rename from ... to
               Action left
               Action comm
            )
         )
      )
      Action comm
   )
   Action comm
)
Got definition: DQueue ::= ( Env H
   ( Env S
      ( compose
         Agent DQ( 0 )
         Agent BUFF
      )
```

```
      Action left
      Action right
      Action shift
   }
   Action left
   Action right
   Action shift
}
Got definition: DQ( _i ) ::= { choice
   { prefix
      { input
         Channel enq
         _x
      }
      { prefix
         { output
            Channel left
            _x
         }
         { prefix
            shift
            Agent DQ( { Operator #0
               _i
               1
            } )
         }
      }
   }
   { prefix
      deq
      { or
         { if ... then
            { Operator #4
               _i
               0
            }
            { prefix
               empty
               Agent DQ( 0 )
            }
         }
         { Fix
            X
            { choice
               { prefix
                  { input
                     Channel right
                     _y
                  }
                  { prefix
                     { output
                        Channel next
                        _y
                     }
                     Agent DQ( { Operator #1
                        _i
                        1
                     } )
                  }
               }
               { prefix
                  shift
                  Agent X
               }
            }
         }
      }
   }
}
Got definition: DCtrl ::= { prefix
   { input
      Channel dci
```

```
      Datumvar start( _i _blk )
   )
   ( prefix
      ( output
         Channel dio
         _blk
      )
      ( prefix
         dint
         ( prefix
            ( output
               Channel dco
               Datum fini( _i _blk )
            )
            Agent DCtrl
         )
      )
   )
)
Got definition: Disk ::= ( prefix
   ( input
      Channel dio
      _blk
   )
   ( prefix
      dint
      Agent Disk
   )
)
Got definition: DSched ::= Agent DS_idle
Got definition: DS_idle ::= ( prefix
   ( input
      Channel ds
      Datumvar req( _cl _blk )
   )
   ( prefix
      ( output
         Channel dci
         Datum start( _cl _blk )
      )
      Agent DS_busy
   )
)
Got definition: DS_busy ::= ( choice
   ( prefix
      ( input
         Channel dco
         Datumvar fini( _cl _blk )
      )
      ( prefix
         ack( _cl )
         Agent DS_check
      )
   )
   ( prefix
      ( input
         Channel ds
         Datumvar req( _cl _blk )
      )
      ( prefix
         ( output
            Channel enq
            Datum req( _cl _blk )
         )
         Agent DS_busy
      )
   )
)
Got definition: DS_check ::= ( prefix
   deq
   ( choice
      ( prefix
```

```
         empty
         Agent DS_idle
      )
      ( prefix
         ( input
            Channel next
            Datumvar req( _cl _blk )
         )
         ( prefix
            ( output
               Channel dci
               Datum start( _cl _blk )
            )
            Agent DS_busy
         )
      )
   )
)
Got definition: DSS ::= ( Env S
   ( compose
      ( Env S
         ( compose
            Agent DSched
            Agent DQueue
         )
         Action enq
         Action deq
         Action next
         Action empty
      )
      ( Env S
         ( compose
            Agent DCtrl
            Agent Disk
         )
         Action dio
         Action dint
      )
   )
   Action dci
   Action dco
)
Got definition: SYS ::= ( Env S
   ( compose
      Agent DSS
      ( compose
         Agent C( 1 )
         Agent C( 2 )
      )
   )
   Action ds
   Action ack( 1 )
   Action ack( 2 )
)
Got definition: C( 1 ) ::= ( prefix
   ( output
      Channel ds
      Datum req( 1 100 )
   )
   ( prefix
      moreone
      ( prefix
         ack( 1 )
         SKIP
      )
   )
)
Got definition: C( 2 ) ::= ( prefix
   ( output
      Channel ds
      Datum req( 2 150 )
   )
```

```
   ( prefix
      moretwo
      ( prefix
         ack( 2 )
         SKIP
      )
   )
)
```

A.3 C++ translation (DSSsim)

```
/*
   Translated by cspt @
   (csp12) DSSsim.csp12 >>> (CSP++) DSSsim.cc
*/

#include "Lit.h"
#include "Agent.h"
#include "Action.h"
#include "main.h"
#include "Listio.c"
#include "List.c"

AGENTDEF( BUFF_, "BUFF", 0 );
AGENTDEF( BUFF_s1, "BUFF", 0 );
AGENTDEF( BUFF_s2, "BUFF", 0 );
AGENTDEF( C_c1, "C", 1 );
AGENTDEF( C_c2, "C", 1 );
AGENTDEF( CELL_, "CELL", 0 );
AGENTDEF( DCtrl_, "DCtrl", 0 );
AGENTDEF( DQ_s1, "DQ", 1 );
AGENTDEF( DQ_v, "DQ", 1 );
   static FreeVar DQ_v__y;
AGENTDEF( DQueue_, "DQueue", 0 );
AGENTDEF( DSS_, "DSS", 0 );
AGENTDEF( DSS_s1, "DSS", 0 );
AGENTDEF( DSS_s2, "DSS", 0 );
AGENTDEF( DS_busy_, "DS_busy", 0 );
AGENTDEF( DS_check_, "DS_check", 0 );
AGENTDEF( DS_idle_, "DS_idle", 0 );
AGENTDEF( DSched_, "DSched", 0 );
AGENTDEF( Disk_, "Disk", 0 );
AGENTDEF( SYS_, "SYS", 0 );
AGENTDEF( SYS_s1, "SYS", 0 );


#ifdef ack_p
 extern ActionProc ack_p;
#else
#define ack_p 0
#endif
 Atomic ack("ack", 1, ack_p);
    static ActionRef ack_r_1( ack, 1  1 );
    static ActionRef ack_r_2( ack, 1  2 );

#ifdef comm_p
 extern ActionProc comm_p;
#else
#define comm_p 0
#endif
 Atomic comm("comm", 0, comm_p);
    static ActionRef comm_r( comm );

#ifdef dci_p
 extern ActionProc dci_p;
#else
#define dci_p 0
#endif
 Channel dci("dci", dci_p);
    static ActionRef dci_r( dci );
```

```
#ifdef dco_p
 extern ActionProc dco_p;
#else
#define dco_p 0
#endif
 Channel dco("dco", dco_p);
    static ActionRef dco_r( dco );

#ifdef deq_p
 extern ActionProc deq_p;
#else
#define deq_p 0
#endif
 Atomic deq("deq", 0, deq_p);
    static ActionRef deq_r( deq );

#ifdef dint_p
 extern ActionProc dint_p;
#else
#define dint_p 0
#endif
 Atomic dint("dint", 0, dint_p);
    static ActionRef dint_r( dint );

#ifdef dio_p
 extern ActionProc dio_p;
#else
#define dio_p 0
#endif
 Channel dio("dio", dio_p);
    static ActionRef dio_r( dio );

#ifdef ds_p
 extern ActionProc ds_p;
#else
#define ds_p 0
#endif
 Channel ds("ds", ds_p);
    static ActionRef ds_r( ds );

#ifdef empty_p
 extern ActionProc empty_p;
#else
#define empty_p 0
#endif
 Atomic empty("empty", 0, empty_p);
    static ActionRef empty_r( empty );

#ifdef enq_p
 extern ActionProc enq_p;
#else
#define enq_p 0
#endif
 Channel enq("enq", enq_p);
    static ActionRef enq_r( enq );

#ifdef left_p
 extern ActionProc left_p;
#else
#define left_p 0
#endif
 Channel left("left", left_p);
    static ActionRef left_r( left );

#ifdef moreone_p
 extern ActionProc moreone_p;
#else
#define moreone_p 0
#endif
 Atomic moreone("moreone", 0, moreone_p);
```

```
#ifdef moretwo_p
 extern ActionProc moretwo_p;
#else
#define moretwo_p 0
#endif
 Atomic moretwo("moretwo", 0, moretwo_p);

#ifdef next_p
 extern ActionProc next_p;
#else
#define next_p 0
#endif
 Channel next("next", next_p);
    static ActionRef next_r( next );

#ifdef right_p
 extern ActionProc right_p;
#else
#define right_p 0
#endif
 Channel right("right", right_p);
    static ActionRef right_r( right );

#ifdef shift_p
 extern ActionProc shift_p;
#else
#define shift_p 0
#endif
 Atomic shift("shift", 0, shift_p);
    static ActionRef shift_r( shift );

DATUMDEF( fini, 2 );
DATUMDEF( req, 2 );
DATUMDEF( start, 2 );
```

```
// % This is the simulated Disk Server, with simulated Disk
and simulated clients.
//
//
// %============================
// % DQueue:  disk request queue
// %
// % Interface:
// %enq!<item>enqueue item
// %deq    dequeue item, followed by:
// %  next?_x  next item returned, or
// %  empty  empty queue indication
// %============================
//
// CELL ::= left?_x -> shift -> right!_x ->CELL.

AGENTPROC( CELL_ )
FreeVar _x;

   left >> _x;
   shift();
   right << _x;
   CHAIN0( CELL_ );
)

//
// % BUFF ::= CELL |> CELL |> CELL .just 2 cells for now
// BUFF ::= (((CELL#{right=comm}) ||
(CELL#{left=comm}))^{comm})\{comm}.

AGENTPROC( BUFF_s1 )

   right_r.rename(comm_r);
   (
      CHAIN0( CELL_ );
   )
```

```
}
AGENTPROC( BUFF_s2 )

   left_r.rename(comm_r);
   {
      CHAIN0( CELL_ );
   }
}
AGENTPROC( BUFF_ )

   comm_r.hide();
   {
      comm_r.sync();
      {
         Agent::compose( 2 );
         Agent* a1 = START0( BUFF_s1, 0 );
         Agent* a2 = START0( BUFF_s2, 1 );
         WAIT( a1 );
         WAIT( a2 );
      }
      Agent::popEnv( 1 );
   }
   Agent::popEnv( 1 );
   END_AGENT;
}

//
// DQueue ::= ((DQ(0) || BUFF)^(left, right, shift))\(left,
right, shift).

AGENTPROC( DQueue_ )

   left_r.hide();
   right_r.hide();
   shift_r.hide();
   {
      left_r.sync();
      right_r.sync();
      shift_r.sync();
      {
         Agent::compose( 2 );
         Agent* a3 = START1( DQ_v, 0, 0 );
         Agent* a4 = START0( BUFF_, 1 );
         WAIT( a3 );
         WAIT( a4 );
      }
      Agent::popEnv( 3 );
   }
   Agent::popEnv( 3 );
   END_AGENT;
}

//
// DQ(_i) ::= enq?_x -> ( left!_x -> shift-> DQ(_i+1) )
//     | deq -> ( (if _i=0 then empty -> DQ(0))
//          + fix X.(right?_y -> ( next!_y -> DQ(_i-1) )
//               | shift -> X)
//          ).

AGENTPROC( DQ_s1 )
#define _i ARG(0)

   Agent::startDChoice( 2 );
      right >> DQ_v__y;
      shift();
   switch ( Agent::whichDChoice() ) {

      case 0:
         next << DQ_v__y;
         CHAIN1( DQ_v, (_i-1) );

      case 1:
```

```
        CHAIN1( DQ_s1, _i );
   }
#undef _i
}
AGENTPROC( DQ_v )
#define _i ARG(0)
FreeVar _x;

   Agent::startDChoice( 2 );
      enq >> _x;
      deq();
   switch ( Agent::whichDChoice() ) {

      case 0:
         left << _x;
         shift();
         CHAIN1( DQ_v, (_i+1) );

      case 1:
         if (_i==0) {
            empty();
            CHAIN1( DQ_v, 0 );
         }
         Agent::compose(1);
         Agent* a5 = START1( DQ_s1, 0, _i );
         WAIT( a5 );
         break;
   }
   END_AGENT;
#undef _i
}

//
// %============================
// % DCtrl:  disk controller
// %
// % Interface:
// %dci!start(_cl, _blk)start operation on block <_blk> for
client <_cl>
// %dco?fini(_cl, _blk)operation finished
// %============================
//
// DCtrl ::= dci?start(_i, _blk)-> dio!_blk-> dint ->
dco!fini(_i, _blk) -> DCtrl.

AGENTPROC( DCtrl_ )
FreeVar _blk;
FreeVar _i;

   dci >> start(_i, _blk);
   dio << _blk;
   dint();
   dco << fini(_i, _blk);
   CHAIN0( DCtrl_ );
}

//
// %============================
// % Disk:  disk drive (simulated)
// %
// % Interface:
// %dio!_blkperform disk i/o on block _blk
// %dint      disk interrupt signalled
// %============================
//
// Disk ::=  dio?_blk -> dint ->Disk.

AGENTPROC( Disk_ )
FreeVar _blk;

   dio >> _blk;
   dint();
```

```
   CHAIN0( Disk_ );
}

//
// %============================
// % DSched:  disk scheduler
// %
// % Interface:
// %ds!req(_cl, _blk)client <_cl> requests operation on block
<_blk>
// %ack(_cl)client's operation finished
// %============================
//
// DSched ::= DS_idle.

AGENTPROC( DSched_ )

   CHAIN0( DS_idle_ );
}


//
// DS_idle ::= ds?req(_cl, _blk) -> dci!start(_cl, _blk) ->
DS_busy.

AGENTPROC( DS_idle_ )
FreeVar _blk;
FreeVar _cl;

   ds >> req(_cl, _blk);
   dci << start(_cl, _blk);
   CHAIN0( DS_busy_ );
}


//
// DS_busy ::= dco?fini(_cl, _blk) -> ( ack(_cl) -> DS_check
)
//      | ds?req(_cl, _blk) -> enq!req(_cl, _blk) -> DS_busy.


AGENTPROC( DS_busy_ )
FreeVar _blk;
FreeVar _cl;

   Agent::startDChoice( 2 );
      DatumVar fini_dv = fini(_cl, _blk);
      dco >> fini_dv;
      DatumVar req_dv = req(_cl, _blk);
      ds >> req_dv;
   switch ( Agent::whichDChoice() ) {

      case 0:
         ack(_cl);
         CHAIN0( DS_check_ );

      case 1:
         enq << req(_cl, _blk);
         CHAIN0( DS_busy_ );
   }
}

//
// DS_check ::= deq -> ( empty -> DS_idle
//          | next?req(_cl, _blk) -> dci!start(_cl, _blk) ->
DS_busy ).

AGENTPROC( DS_check_ )
FreeVar _blk;
FreeVar _cl;

   deq();
   Agent::startDChoice( 2 );
      empty();
      DatumVar req_dv = req(_cl, _blk);
      next >> req_dv;
```

```
   switch ( Agent::whichDChoice() ) {

      case 0:
         CHAIN0( DS_idle_ );

      case 1:
         dci << start(_cl, _blk);
         CHAIN0( DS_busy_ );
   }
}

// %============================
// % DSS:  disk server subsystem
// %
// % Interface: (see DSched)
// %============================
//
// DSS ::= (  (DSched || DQueue)^(enq,deq,next,empty)
//   ||
//     (DCtrl || Disk)^(dio,dint)   )^(dci,dco).

AGENTPROC( DSS_s1 )

   enq_r.sync();
   deq_r.sync();
   next_r.sync();
   empty_r.sync();
   {
      Agent::compose( 2 );
      Agent* a6 = START0( DSched_, 0 );
      Agent* a7 = START0( DQueue_, 1 );
      WAIT( a6 );
      WAIT( a7 );
   }
   Agent::popEnv( 4 );
   END_AGENT;
}
AGENTPROC( DSS_s2 )

   dio_r.sync();
   dint_r.sync();
   {
      Agent::compose( 2 );
      Agent* a8 = START0( DCtrl_, 0 );
      Agent* a9 = START0( Disk_, 1 );
      WAIT( a8 );
      WAIT( a9 );
   }
   Agent::popEnv( 2 );
   END_AGENT;
}
AGENTPROC( DSS_ )

   dci_r.sync();
   dco_r.sync();
   {
      Agent::compose( 2 );
      Agent* a10 = START0( DSS_s1, 0 );
      Agent* a11 = START0( DSS_s2, 1 );
      WAIT( a10 );
      WAIT( a11 );
   }
   Agent::popEnv( 2 );
   END_AGENT;
}

//
// SYS ::= (DSS || (C(1)|||C(2)) )^(ds,ack(1),ack(2)).

AGENTPROC( SYS_s1 )

   Agent::compose( 2 );
```

```
   Agent* a12 = START1( C_c1, 0, 1 );
   Agent* a13 = START1( C_c2, 1, 2 );
   WAIT( a12 );
   WAIT( a13 );
   END_AGENT;
}
AGENTPROC( SYS_ )

   ds_r.sync();
   ack_r_1.sync();
   ack_r_2.sync();
   {
      Agent::compose( 2 );
      Agent* a14 = START0( DSS_, 0 );
      Agent* a15 = START0( SYS_s1, 1 );
      WAIT( a14 );
      WAIT( a15 );
   }
   Agent::popEnv( 3 );
   END_AGENT;
}

//
// %============================
// % Demo
// %============================
//
// C(1) ::= ds!req(1,100)->moreone->ack(1)->SKIP.

AGENTPROC( C_c1 )

   ds << req(1, 100);
   moreone();
   ack(1);
   END_AGENT;
}

//
// C(2) ::= ds!req(2,150)->moretwo->ack(2)->SKIP.

AGENTPROC( C_c2 )

   ds << req(2, 150);
   moretwo();
   ack(2);
   END_AGENT;
}

//
//


main( int argc, char* argv[] )
{
#ifndef START
#define START SYS_
#endif
    MAIN( argc, argv, START );
}
```

A.4 Simulated Disk removed (DSS)

```
% This is the "real" Disk Server, with simulated clients.
The Disk
% process is commented out in DSS.  Note, it does no harm to
leave the
% Disk process defined.

%============================
% DSS:  disk server subsystem
%
```

```
% Interface: (see DSched)
%============================

%DSS ::= (  (DSched || DQueue)^(enq,deq,next,empty)
%    ||
%      (DCtrl || Disk)^(dio,dint)    )^(dci,dco).

DSS ::= (  (DSched || DQueue)^(enq,deq,next,empty)
     ||
       DCtrl    )^(dci,dco).
```

A.5 External routines (DiskProcs.cc)

```
/*
 * DiskProcs.cc
 *
 * External action procedures linked to Channel dio and
Atomic dint
 */

#include "Lit.h"
#include "Action.h"


/*
 * Replaces dio?block:  start I/O on given block #
 */
void dio_chanInput( ActionType t, ActionRef* a, Var* v, Lit*
block )
{
    cerr << "*** dio_chanInput: Starting I/O on block # " <<
*block << endl;
}

/*
 * Replaces dint:  disk interrupt
 */
void dint_atomic( ActionType t, ActionRef* a, Var* v, Lit* l
)
{
    cerr << "*** dint_atomic: Receiving disk interrupt" <<
endl;
}
```

A.6 Execution trace (DSS)

```
|=DS_idle [ds$req( 1, 100 )]
|=C( 1 ) [moreone]
Action: moreone
*** dio_chanInput: Starting I/O on block # 100
|=DCtrl [dio$100]
*** dint_atomic: Receiving disk interrupt
|=DCtrl [dint]
|=DS_idle [dci$start( 1, 100 )]
|=DS_busy [dco$fini( 1, 100 )]
|=DS_busy [ack( 1 )]
|=DQ( 0 ) [deq]
|=DQ( 0 ) [empty]
|=DS_idle [ds$req( 2, 150 )]
|=C( 2 ) [moretwo]
Action: moretwo
*** dio_chanInput: Starting I/O on block # 150
|=DCtrl [dio$150]
*** dint_atomic: Receiving disk interrupt
|=DCtrl [dint]
|=DS_idle [dci$start( 2, 150 )]
|=DS_busy [dco$fini( 2, 150 )]
|=DS_busy [ack( 2 )]
|=DQ( 0 ) [deq]
```

```
|=DQ( 0 ) [empty]
idletask: All tasks IDLE!
Agent::exit_fn: Dump printed on stdout

== AGENT DUMP ==

Current # Literals: 1; High water marks:  9 Literals, 13
tasks
================================================================
task #12 'CELL' (IDLE)
This task running as Agent CELL
My sync flag is #1 (LSB=#0) in [10] -- waiting for sync on
comm
================================================================
task #8 'CELL' (IDLE)
This task running as Agent CELL
My sync flag is #1 (LSB=#0) in [10] -- waiting for sync on
left
================================================================
task #11 'BUFF' (IDLE)
This task running as Agent BUFF
================================================================
task #6 'DQ' (IDLE)
This task running as Agent DQ( 0 )
 Waiting for sync on enq
My sync flag is #1 (LSB=#0) in [10]
 Waiting for sync on deq
My sync flag is #1 (LSB=#0) in [10]
================================================================
task #10 'DQueue' (IDLE)
This task running as Agent DQueue
================================================================
task #9 'DS_idle' (IDLE)
This task running as Agent DS_idle
My sync flag is #0 (LSB=#0) in [01] -- waiting for sync on ds
================================================================
task #7 'DCtrl' (IDLE)
This task running as Agent DCtrl
My sync flag is #1 (LSB=#0) in [10] -- waiting for sync on
dci
================================================================
task #5 'DSS' (IDLE)
This task running as Agent DSS
================================================================
task #4 'SYS' (TERMINATED)
 Result = 0
================================================================
task #3 'DSS' (IDLE)
This task running as Agent DSS
================================================================
task #2 'idle' (RUNNING)
================================================================
task #1 'SYS' (IDLE)
This task running as Agent SYS
================================================================
task #0 'main' (IDLE)
task::stop exiting
```

APPENDIX B

CSP++ User's Manual

B.1 Compiling the framework and translator

Since the platform for the AT&T cfront version, CSP++ V1.0, is essentially obsolete, we only give directions for compiling the Pthreads version. If the former version is of interest, its makefile should be consulted.

B.1.1 Source distribution

Figure 11 shows the directory structure that results from unzipping the source archive of V2.1. Rectangles represent subdirectories. It is assumed that g++ is available.

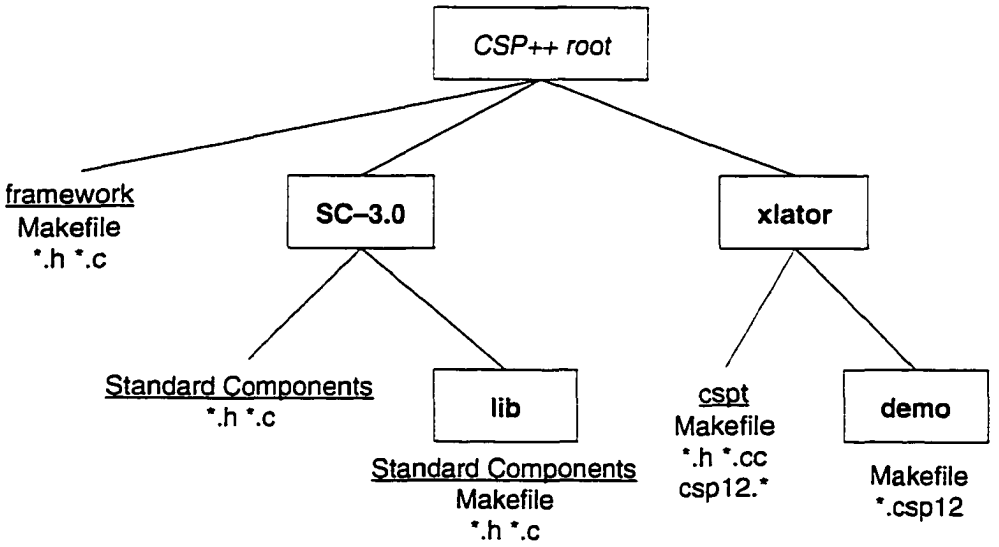


Figure 11: CSP++ V2.1 source code organization

One should start by compiling the USL Standard Components library: "make" SC-3.0/lib/Makefile. The other files in the SC-3.0 subdirectory are for inclusion at com-

pile time.[1]

Next, compile the CSP++ framework using the Makefile in the top-level directory. This will result in a number of *.o object files that can be linked with a translated application later.

Compile the translator using xlator/Makefile. This will run flex and bison, so their pathnames may need to be modified to suit a given installation, not only for the binaries, but also for the flex library (libfl.a).

To test the software, try compiling the example programs in the demo subdirectory using the Makefile provided. The translator will be run on the *.csp12 specifications, then the resulting *.cc files compiled. These will link with the framework elements in the top-level directory. The Makefile assumes that all the Pthreads routines are in the libpthread.so ("-lpthread"). If this is not the case—for example, on our Solaris host it was also necessary to also load libposix4.so ("-lposix4")—unsatisfied externals will occur at link time, and the Makefile should be modified accordingly. Run the examples by typing "DSS" or "DSSsim" with various command line options (Section B.4).

---

1. These files were designed to be compiled with cfront, which builds a precompiled template repository. Complex methods, considered too lengthy to go in, say, a foo.h, were placed in an additional foo.c file where cfront knew to look for them. (The default extension for C++ files was .c, not .cc.) But for g++ compilations, whenever foo.h is to be included by a .c file, one must also explicitly include foo.c in order to get the rest of the template methods. Failure to do so results in mysterious load-time errors, because those methods will never have been created in any object file.

This scenario is made more confusing by the fact that in a Standard Components "lib" subdirectory there might be found yet another foo.c file; that is, having the same name as the file just described, but with different contents. These .c files were supposed to be separately compiled and stored in a library archive to supply some non-parameterized template methods. We compile these into foo.o files, and explicitly load them with the rest of the object files that make up a CSP++ executable program.

B.1.2 Compile-time symbols

In some cases, a user may wish to have greater visibility into the inner workings of the CSP++ run-time framework than is provided by tracing. This can be obtained by recompiling the framework with certain preprocessor symbols defined:

1. MEMWATCH: Orders the `Literal` routines to print all storage-related activities, including allocating, assigning, and deleting, in order to check for leaks. Also enables code in `Lit::memStatus` to print an annotated message giving the current number of outstanding `Literals`.
2. ACTWATCH: Orders details printed of every action's execution, including step-by-step environment stack search.

In the top-level Makefile, one could specify, for example, `OPTS="-DACTWATCH"` to get action logging. The output of these options can be quite voluminous.

B.2 Running the cspt translator

The syntax for invoking the translator is as follows:

```
cspt [-s] [-d] [-t] <csp12 file>
```

If the input file has a .csp12 extension, C++ output is produced in a similarly-named file with .cc extension, otherwise .cc is just appended to the input file name.

Here are the command line options:

1. **-s**: copy the csp12 source statements into the translated output file, interleaved so that they appear just prior to their translated C++ code.
2. **-d**: "debug" option, produces a syntax tree on `cerr` (stderr).

3. **-t**: turns on the flex and bison trace features, which logs how each csp12 statement is analyzed and parsed.

Note that error recovery is primitive. If the translator encounters an error in the input, it will generally report the problem and abort processing without attempting to read further.

B.3 Compiling the synthesized code

When g++ is invoked to compile the output of the translator, it must have access to the .h include files in the top-level directory, and those in the SC-3.0 subdirectory. The compiler's "-g" option may be used to insert information for symbolic debugging.

The "-D*symbol=value*" option is used for two purposes:

1. Specify an external routine of name *value* to be linked to an action. If the action (atomic or channel) is named `foo`, *symbol* should be written with the suffix "_p", i.e., `foo_p`. Any number of "-D" definitions may be supplied.

2. Override the normal starting point for execution of the compiled system, which is the agent named `SYS`. If the specification does not contain such an agent, or if it is desired to start execution elsewhere, redefine the START symbol as follows:

   `-DSTART=agentproc`

   Note that *agentproc* is not the csp12 name of the agent, like E(2), but rather the name given by the translator to its corresponding procedure, e.g., `E_c2`. These names are most easily located in the .cc file as arguments to the AGENTPROC macro.

g++ can also be used to link/load the compiled system with any external routines.

From the top-level directory load: Lit.o, Agent.o, Action.o, and task.o. From SC-3.0/lib load: Bits.o, List.o, and Pool.o. Specify library -lpthread (and -lposix4 if needed).

B.4 Invoking the compiled system

These option flags should be typed with the command that executes your CSP++ system, i.e., the binary file produced from the previous link/load step:

1. **-t**: Print traces on `cerr` (stderr) of every action taken.
2. **-i**: Starts an idler task which wakes up periodically and checks the status of all non-terminated tasks. If all are found to be idle, a dump of the task status is performed, and the program exits. High water marks are reported for the maximum number of `Literals` and tasks in existence.
3. **-q**: Exits with a fast "quit" to the command line, by suppressing the usual dump, when STOP is executed or all tasks are idle (-i option).

The idler task interval is hardcoded at 2 seconds. This can be modified in main.h (see argument to `idletask` constructor). The dump and termination can also be triggered by an agent's executing STOP. Suppression of the dump (-q option) is useful when making timing measurements on an executable.

APPENDIX C

Restrictions and Limitations

C.1 Restrictions

Some restrictions are listed in Table 5. Most of them are in terms of constructs which may

AREA	RESTRICTION	CONSEQUENCE
Action	If an external routine is linked to an Action, the Action cannot also be used for sync.	Make sure that internal and external Actions are distinguished in the CSP specification.
Agent	All definitions of the same-named agent must have the same number of non-overlapping arguments. Constant arguments can only be integers.	One cannot define, say, both X and X(i), nor X(0) and X(i). Instead, define only X(i) and start by testing i for 0.
Atomic	Subscripts must be integers.	"P::=a(foo(1))→Q" is illegal since the Datum foo is considered a subscript. To communicate a Datum, use a channel instead.
Channel	Default channel input action is to accept an integer.	Default channel input actions cannot be used for typing in non-Num data values.
Datum	A given DatumID must always appear with the same number of subscripts, because Datums have global scope.	One cannot write, say, foo(1,2) in one agent body and foo(x) in another.

Table 5: Restrictions in current CSP++

be allowed in csp12 (or are loosely specified), but which were not carried over into CSP++. None would be considered difficult to live with.

C.2 Numerical limitations

There are a handful of compile-time constants, listed in Table 6, which serve as fixed array dimensions. There are also limits on function argument lists. All such limits could have been avoided by certain techniques (such as variable-length argument lists), but this

CONSTANT	LIMITATION (Max. no.)	IMPACT OF INCREASING
AG_ARGS	`AgentProc` arguments (4)	More storage for array of `Lits`, and more calls to `Lit` constructors/destructors when `Agents` start/ terminate.
AG_COMPOSE	Agents that can be composed (8)	More storage for `syncFlags` bit strings in `EnvSync` objects (negligible).
AT_SUBS	`Atomic` subscripts (4)	Code more arguments for `ActionRef` constructor and `Atomic::operator()`.
none, see Lit.h	`Datum` subscripts (4)	Code more DATUM_*n* macros.

Table 6: Locations of limitations

would have disabled argument type checking and was not considered worth the trouble. These constants were set to arbitrary useful values, and can all be increased by recompiling CSP++. In some cases, more code must be written. Originally, these constants were in several .h include files, but as of V2.1 they are all in Limits.h.

APPENDIX D

Detailed Design of cspt Translator

Since CSP++ was purposely designed to make the job of translating from CSP straightforward, it is natural that the translator should not be a greatly complex piece of software. It was built using the conventional compiler-writing tools, **LEX** and **YACC**, or actually their Gnu cousins, **flex** [Paxs95] and **bison** [Donn92], that come with Red Hat Linux.

In the sections below we first present a brief overview, followed by a detailed description of the translator's two phases. That complete, we return once more to our disk server case study to show its translation and execution.

D.1 Overview

Cspt operates in two phases: first, a combined lexical and syntax phase which scans the csp12 input file and produces a syntax tree; second, a code generation phase that walks the tree and produces a C++ output file. In addition to the syntax tree, the other data structures that persist between phases are the symbol tables.

Object-oriented design has been used throughout. The syntax tree is built from `ParseNode` objects, each of which knows how to generate itself and its subtree in the code generation phase. The C++ STL has been utilized wherever possible to avoid writing and debugging code for stock data structures. Its benefits have been especially appreciated in the areas of tree building, tree-navigation, and symbol table access.

In contrast to the run-time framework, where pains were taken to bolster efficient execution, no such design goals were applied to the translator, either with regards to memory use or execution speed. The sole criteria were a logical design and ease of coding. Despite this "spendthrift" policy vis-à-vis hardware resources, translating the case study on a Pentium 200 running Linux took well under one second.

The only technical challenges to speak of arose in three areas:

1. identifying and extracting complex subagents for separate generation, while ensuring access to their parents' symbols
2. managing the symbols for multiple agent definitions and binding them to agent invocations
3. handling agent termination, whether by chaining, returning, or starting new agents, depending on the context

None of these were at all intractable, though the third one, intuitive enough when hand translating, proved to be surprisingly resistant to being reduced to a deterministic decision algorithm.

Some attention has been given to diagnosing problems in the translator's input, down to the offending line and character where practical. Still, this feature is fairly rudimentary at present. Cspt also has been equipped with command line switches that are used to turn on flex and bison debugging features, and to produce translated output interleaved with csp12 source statements (in the form of C++ comments). These options are described in the translator's User's Manual (see Appendix B.2).

D.2 Lexical and syntax phase

First, a review of how flex and bison are used to process a language:

Translation is driven by the bison-generated parser, which is invoked from the user's `main()` function (in bison source file csp12.y). The parser shifts input tokens on and off its stack until it recognizes patterns in the language's grammar. The patterns are supplied by the compiler writer in pseudo-BNF style when the bison generator is invoked. The parser, in turn, gets its input tokens from the flex-generated scanner, which breaks them out of the input stream according to the language's lexical features. Those rules are also supplied by the compiler writer, along with a routine to read the actual input file and fill the scanner's buffer. When the parser recognizes a grammatical pattern, it calls a user routine, which in our case adds nodes to the parse tree under construction.

In this fashion, the parser and scanner carry on in tandem until the csp12 source file is fully processed. This constitutes the first phase of cspt.

The rules used to customize flex and bison are given below. This is followed by a detailed description of the relevant data structures, the parse tree and the symbol tables.

D.2.1 Lexical rules

These rules are contained in the flex source file csp12.lex. First are the csp12 operators and delimiters. These range from single-character tokens such as "!" and "#" to double- and triple-character tokens "->", "| |", "::=", "| | |", etc. A handful of reserved words are recognized by flex: `done`, `if`, `fix`, `SKIP`, `STOP`, and `then`. Such tokens are reported to the parser by code number. For single-character tokens, the ASCII value of the character is its number. The others are assigned code numbers from a table in the file csp12.tab.h,

which is generated by bison from the `%token` statements appearing in the bison source file csp12.y.

The above description applies to tokens that are fixed character strings. Other kinds of tokens, such as numbers and identifiers, are recognized by patterns. Csp12 uses four types:

1. Lower-case identifier or `LID`, used for action and datum names
2. Upper-case identifier or `UID`, used for agent names
3. Variable or `VAR`, an identifier starting with underscore "_"
4. Numeric or `NUM`, integers

`NUM` tokens are reported to the parser using their integer value.

Identifiers can contain any alphanumeric character, plus "_" and "'". The types are distinguished based on the leading character: upper case, lower case, or underscore. They are stored as C++ `string` objects, the instances being allocated by the scanner. The value of a `LID`, `UID`, or `VAR` passed to the parser is its `string*` pointer. No attempt is made to economize on storage by recognizing identical previously-allocated strings and copying their pointers.

The scanner is also told to recognize csp12 style line-oriented comments: anything to the right of "%" is ignored.

D.2.2 Grammar rules

These rules are contained in the C++ source file csp12.y. They are listed in Table 7, rewritten in conventional BNF. Thus, Table 7 helpfully documents both the accepted

Accepted csp12 syntax in BNF	Parent PNtok	Parent PNcop	Parent PNcid	Subclass Name	Pseudocode[a] ctor[b]	Pseudocode[a] prep()	Pseudocode[a] gen() and details for entries marked ">" (ctor = constructor)
	ParseNode				>	OK	ctor: store line number gen(): OK
	*	PNtok			{ }	-	-
		*	PNcop		{ }	apply prep/gen to each operand in turn; stop on bad status	
			*	PNcid	{ }	>	prep(): apply to each arg/subscript; stop on bad status gen(): output name
<definition> ::= <signature> '::=' <agent> '.'		*		PNdefn	{ }	NC	prep signature and agent; use agent's symbol entry to gen AGENTPROC, arg #defines, and FreeVars (genAgentProc); gen agent body; "ENDAGENT" if needed; gen arg #undefs (genEndAgent)
<agent> ::= ('(' <agent> ')' | <prefix>	*see <prefix> below*						
| <prefix> 'I' <prefix> { 'I' <prefix> }		*		PNchoice	{ }	-	"Agent::startDChoice(*n*)"; set flag for PNinput (DatumVar gen); genPre actions; "Agent::whichDChoice()"; genPost agents
| FIX <UID> '.' <agent>		*		PNfix	{ }	>	prep(): use agent's symbol entry to extract agent as subagent (makeSubAgent); change <UID> refs in subagent to new PNconstSub (changeConstRefs); gen subagent gen(): gen the PNconstSub
| <agent> ';' <agent> { ';' <agent> }		*		PNseq	{ }	-	gen each agent, flagging last one
| '@' <agent>		*		PNloop	{ }	-	"while(1) {"; gen agent; "}"

Table 7: BNF syntax with corresponding parse node classes

Accepted csp12 syntax in BNF	Parent PNtok	Parent PNcop	Parent PNcid	Subclass Name	Pseudocode[a] ctor[b]	Pseudocode[a] prep()	Pseudocode[a] gen() and details for entries marked ">" (ctor = constructor)
| <agent> '|' <agent> | <agent> '||' <agent>		*		PNcompose	{ }	>	prep(): prep simple agents; complex: use agent's symbol entry to extract subagents (makeSubAgent), then gen gen(): "Agent::compose(*n*)"; "START" each agent; "WAIT" each agent
| <agent> '^' '{' <name>{,<name>} '}' | <agent> '\' '{' <name>{,<name>} '}' | <agent> '#' '{' <rename>{,<rename>} '}'		*		PNenv	{ }	-	gen the ActionRefs; "."; "sync()", "hide()", or gen PNrename; gen the associated agent; "Agent::popEnv(*n*)" if needed
| <agent> '+' <agent>		*		PNor	{ }	-	gen each agent
| STOP	*			PNstop	{ }	-	"Agent::stop()"
| SKIP	*			PNskip	{ }	-	set flag to get ENDAGENT generated
| <UID>| '(' <exp>{,<exp>} ')']			*	PNconst	{ }	>	prep(): find in agentTable, get agentproc name via bindSig(*args*) gen(): "CHAIN", "START", or "START/WAIT" depending on context
| IF <exp> THEN <agent>)		*		PNifthen	{ }	>	prep(): prep agent gen(): "if ("; gen exp; ") {"; gen agent; "}"
<prefix> ::= <action> '->' <agent>		*		PNprefix	{ }	-	gen(): - genPre(): gen action genPost(): gen agent

Table 7: BNF syntax with corresponding parse node classes (Continued)

Accepted csp12 syntax in BNF	Parent			Subclass Name	Pseudocode[a]		
	PNtok	PNcop	PNcid		ctor[b]	prep()	gen() and details for entries marked ">" (ctor = constructor)
<signature> ::= <UID> '(' <numvar>{,<numvar>} ')'			*	PNsig	>	>	ctor: find in agentTable, or insert new variant prep(): find signature in agentTable, set its symbol entry as the translation context; setup symbol entry to handle symbols for variant (prep) gen(): NC
<numvar> ::= (<NUM>	*			PNnum	{ }	-	output value
| <VAR>)	*			PNvar	{ }	>	prep(): report to agent's symbol entry (addvar) with "global" flag if in subagent gen(): output var name, maybe globalized, obtained from agent's symbol entry (ref)
<action> ::= (DONE	*			PNdone	{ }	-	-
| <LID>| '(' <exp>{,<exp>} ')']			*	PNatomic	>	OK	ctor: find/insert in actionTable gen(): output name, gen subscripts
| <LID> '?' (<VAR> | <datumvar>)		*		PNinput	>	-	ctor: new PNchannel gen(): if datumvar, "DatumVar" *temp* "=" gen datumvar; gen PNchannel; ">>"; gen PNvar or *temp*
| <LID> '!' <exp>)		*		PNoutput	>	OK	ctor: new PNchannel gen(): gen PNchannel; "<< ("; gen exp; ")"
	*			PNchannel	>	-	ctor: find/insert in actionTable gen(): output name
<datumvar> ::= <LID>| '(' <VAR>{,<VAR>} ')']			*	PNdatumvar	>	-	ctor: find/insert in datumTable gen(): output name, gen subscripts
<name> ::= <LID>| '(' <NUM>{,<NUM>} ')']			*	PNaction	{ }	OK	find in actionTable, output ActionRef, gen subscripts

Table 7: BNF syntax with corresponding parse node classes (Continued)

Accepted csp12 syntax in BNF	Parent: PNtok	Parent: PNcop	Parent: PNcid	Subclass Name	Pseudocode[a]: ctor[b]	Pseudocode[a]: prep()	Pseudocode[a]: gen() and details for entries marked ">" (ctor = constructor)
<rename> ::= <name> '=' <name>		*		PNrename	>	OK	ctor: get 2nd name into actionTable (makeAtomic) gen(): gen 1st PNaction; ".rename("; gen 2nd PNaction; ")"
<exp> ::= (<numvar>	*see <numvar> above*						
| <LID> '(' <exp>{,<exp>} ')'			*	PNdatum	>	OK	ctor: find/insert in datumTable gen(): output name, gen subscripts
| '–' <exp> | <exp> <op> <exp>) <op> ::= ('+' | '–' | '*' | '/' | '=' | '<' | '>' | '=<' | '=>' | '<>')		*		PNop	{ }	OK	"("; gen left exp; *op*; gen right exp; ")"
Prep-time node substitution: new extracted subagent's <signature>			*	PNsigSub	{ }	>	prep(): note subagent no. in translation context gen(): NC
replaces complex <agent> subtree, refers to subagent	PNconst			PNconstSub	{ }	OK	default to PNconst::gen()

Table 7: BNF syntax with corresponding parse node classes (Continued)

a. Abbreviations: { } = no-op; - = default to parent's method; OK = no-op, return good status (0); NC = method is not called; "foo" = output "foo"

b. Constructor: The obvious action of storing arguments in data members is not explicitly written out.

input syntax and the translator's related construction side-by-side in one place. It should be the starting place for anyone wishing to modify the translator.

There is an area of significant divergence from csp12 syntax. It will be recalled that csp12 is a tool for model checking as well as simulation. Thus it includes syntactic elements for collecting traces and performing logical operations such as temporal verification. That subset of csp12, which is not synthesizable, is not supported by cspt.

D.2.3 Parse tree

Without exception, bison rule recognition always results in either creating a new `ParseNode`, or adding a token to an operand list in preparation for creating a `ParseNode` from the list. An important typedef is `LOPNP`, an acronym for "list of `ParseNode` pointers." `LOPNP` is defined as the STL template `deque<ParseNode*>`, and manipulated with the STL's container class methods.

The `ParseNode` class hierarchy (ParseNode.h) is shown in Figure 12. The abstract base class records the csp12 input file line number associated with the node (used to print diagnostics for errors discovered in phase two). The virtual `prep()` and `gen()` methods (explained under Section D.3 "Code generation phase") specify the output stream to print to, and provide a status return which can be used to abort the translation.

The subclasses define three broad categories of specific parse node types:

- `PNcop`: complex operators, having a list of operands (`pnl`)
- `PNtok`: simple tokens
- `PNcid`: complex identifiers, having a name and a list of arguments or subscripts

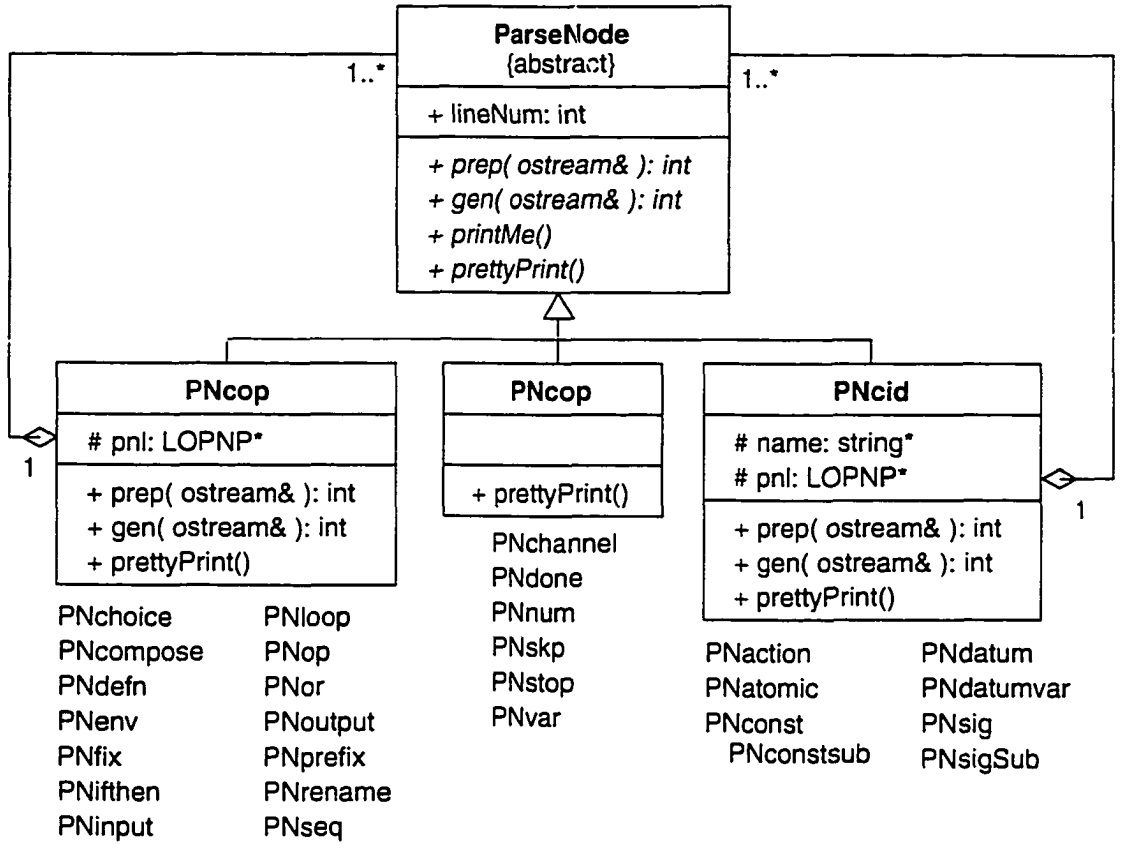


Figure 12: Parse node class hierarchy

These all have default `prep()` and `gen()` methods, which may or may not suit their subclasses. If not, the methods can be overridden.

The specialized parse node subclasses (ParseNodes.h) are listed in Figure 12 underneath their respective base classes. The inheritance goes down just this additional level, with the exception of `PNconst`, which is the parent of `PNconstSub`. These classes are listed opposite their corresponding BNF statements in Table 7. Their constructors (abbreviated "ctor" in the table), in most cases, simply store their arguments, though some perform symbol table lookups (see next section).

For debugging purposes, virtual methods are provided for printing each parsed defi-

nition. When the "-d" debug flag is used on the cspt command line (see Appendix B.2), `prettyPrint()` is called each time an agent definition is recognized. Complex nodes print their name or type and then recursively `prettyPrint()` each of their operands. Simple nodes print their name or value. The result is a neatly indented, nested representation of the parse tree printed on `cerr` (`stderr`). A sample is given in Appendix A.2.

Not shown in Figure 12 are a handful of additional virtual methods used for querying node types and values: `isNum()`, `intVal()`, `getName()`, and the like. The abstract base class supplies default methods for these (always returning false or 0); relevant subclasses simply override these defaults.

D.2.4 Symbol tables

Symbol collection is an important task of the translator's first phase. A number of `ParseNode` subclass constructors require access to symbol tables: storing new names and looking up existing ones.

Symbol tables are constructed with the help of the STL `map` template. The `SymTable<T>` class (Symbols.h), derived from `map<string*,T*,SYcompare>`, sets up a mapping from identifiers (`string*`) to symbol entries (`T*`). The object `SYcompare`[1] provides the operator needed to order any pair of `string*` identifiers by invoking the C++ `string::compare()` function. The underlying `map` functionality

1. "Compare objects" are a syntactically obscure aspect of the STL (see [Aust99]). They should provide a function call operator that returns true if the first operand is "less than" (however one cares to define that) the second operand. We use lexicographic string comparison:

```
struct SYcompare {
   bool operator()( string* s1, string* s2 ) const
   { return ( s1->compare( *s2 ) < 0 ); }
};
```

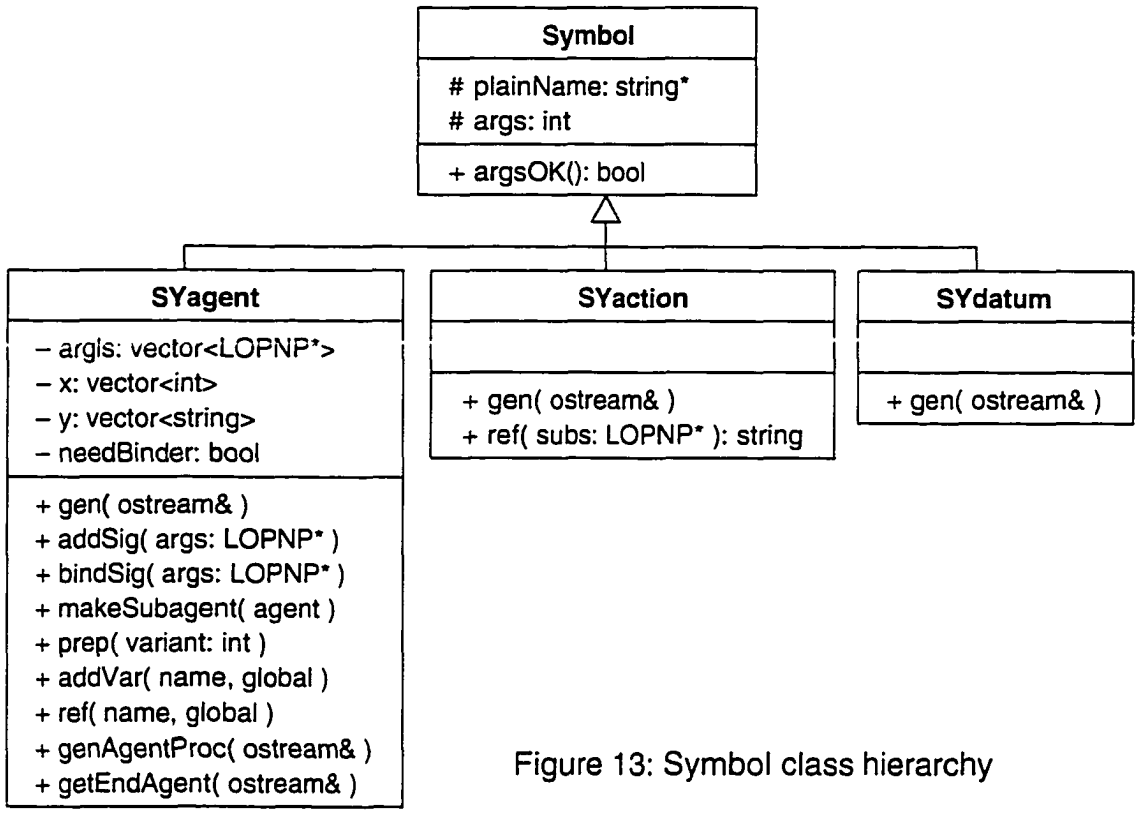


Figure 13: Symbol class hierarchy

ensures that there are no duplicate definitions in a `SymTable` object, and allows its symbols to be iterated in lexicographic order.

Different kinds of symbol entries are defined for agents, actions, and datums. The hierarchy for the symbol entry classes is shown in Figure 13. The three global symbol tables are then defined thusly:

```
SymTable<SYagent> agentTable;
SymTable<SYaction> actionTable;
SymTable<SYdatum> datumTable;
```

The only method these three tables, or more precisely, the symbol entry classes, have in common is the `gen()` method, used to output symbol definitions in the code generation phase. Other features of these classes are described in the following three sub-

sections.

D.2.4.1 SYagent entry

One `SYagent` object is created to record all *variants* of a particular agent name; e.g., `SYS(1,1)`, `SYS(2,_i)`, and so on, are variants of the name "SYS". Once an agent name has been defined, it sets the pattern regarding arguments, and all subsequent variants must have the same argument cardinality.

As each variant is encountered in the input stream, it is processed into the same-named symbol entry by the `addSig(`*args*`)` method:

- Its argument list is appended to the `argls` vector.
- The arguments are analyzed, resulting in its signature and appropriate entries being appended to the `y` and `x` vectors, respectively.

These data members are later used for agent binding via the `bindSig(`*args*`)` method. If compile-time binding ever fails, the `needBinder` flag gets set, which will cause an `AgentBinder` to be dumped out when `gen()` is later invoked on the symbol entry.

When the code generation phase commences, subagents may be extracted. In that case a special `PNdefn` node needs to be created containing the extracted subtree. This chore is performed by the `makeSubagent(`*agent*`)` method.

The *agent* argument (that is, the subtree) gets replaced by a `PNconstSub` node referring to the subagent name. Such names are assigned sequential numbers within the symbol entry (e.g., `SYS_s1`, `SYS_s2`, etc.). The subtree itself is reinstalled under a new `PNdefn` node, along with a `PNsigSub` signature node containing the subagent's number. When `gen()` is invoked, definitions for all subagents are output.

Because of the possibility of subagent extraction, the `SYagent` class must provide facilities for collecting a variant's variables and converting them to global scope if they are referenced in a subagent. This symbol-table-within-a-symbol-table is implemented by three methods:

- `prep(`*variant*`)` : sets up the symbol entry so that `prep()` and `gen()` will utilize symbols for the designated variant number.
- `addVar(`*name*`,` *global*`)` : invoked whenever a variable name is encountered in a channel input context during the `prep()` subphase of code generation.
- `ref(`*name*`,` *global*`)` : obtains the C++ name of the variable, which gets "uniquified" if the variable is used in subagents. `ref()` detects the error of using the value of a variable before setting it (`ref()` without a prior `addVar()`).

Two more methods provide the start- and end-of-code-block generators for the variant specified by `prep(`*variant*`)`. These are `genAgentProc(ostream&)` and `genEndAgent(ostream&)`.

D.2.4.2 SYaction entry

One `SYaction` object is created for each atomic action or channel. Atomics are subject to a limited variant phenomenon in that if they are subscripted, all subscripts actually used must be recorded so that `ActionRef` objects can be output by `gen()`. The `ref(`*subs*`)` method serves this purpose, by both recording the subscripts and also returning the C++ name of the corresponding `ActionRef`.

Aside from `ActionRefs`, `gen()` also outputs `Channel` and `Atomic` definitions. Each definition is preceded by a block of preprocessor code that tests whether the user has provided a compile-time definition of the symbol *action*`_p`. If so, the symbol's

value is used as the name of an external `ActionProc` to be linked with the action. If not, the default value of zero suppresses external linkage.

D.2.4.3 SYdatum entry

One `SYdatum` object is created for each unique `Datum` or `DatumVar` name. The first occurrence fixes the number of subscripts, and subsequent occurrences are validated via the `Symbol::argsOK()` method. Invoking `gen()` outputs a `DATUMDEF` macro with the appropriate number of subscripts.

D.3 Code generation phase

When we arrive at the code generation phase in the `main()` function of csp12.y, the parse tree is complete and all symbols have been collected. The symbol tables are not, however, in their final forms, since the subsequent extraction of subagents may cause some variables to be globalized (names changed), and of course new subagent names to be added, as described in Section D.2.4.1 above.

Code generation has two subphases:

1. Generate the parse tree to a scratch file.
2. Generate all the symbol definitions to the output file.

This is followed by copying the scratch file to the output file to complete the translation.

Generating the tree is an alternating two-step process, consisting of a `prep()` step followed by a `gen()` step. The purpose of the prep is threefold:

1. to extract subagents where required (`PNcompose` and `PNfix`)
2. to note the occurrence and scope of input variables (`PNvar`)

3. to bind agent constants to agentproc signatures (`PNconst`)

In general, calls to `prep()` are simply relayed down the tree by complex nodes to leaf nodes. However, if a node type knows that no candidates can lie below it, it can limit descent by returning a good status (viz "OK" entries in Table 7).

Subagent extraction is needed whenever a complex agent expression is specified where a simple operand is required. In such cases, the `prep()` method extracts the subtree as a new agent, invokes `prep()` and `gen()` on it, so that it physically appears in the translated output separate from its parent, and then substitutes the name of the extracted subagent for the subtree. This process of extraction can recurse as deeply as necessary, resulting in a series of nested `prep()` calls and finally a `gen()` at the lowest level, and so on back up the tree.

The parse tree at its base is simply an `LOPNP`, having one `PNdefn` node for each statement in the CSP specification. Generation involves calling `gen()` on each `PNdefn` node, and checking the return code to see if an error has occurred. Thus, `PNdefn::gen()` can be regarded as "translation headquarters" at the agent definition level. Table 7 on page 148 gives the pseudocode for each parse node, showing what their `prep()` and `gen()` methods do. In many cases, defaulting to the parent's methods is sufficient. Error detection takes place during this phase, but regrettably, error *recovery* is essentially nonexistent: most errors print a diagnostic and abort the translation without attempting to carry on any further.

Following generation of the syntax tree to the scratch file, the symbol definitions, now in final form, are generated to the output file. First, there are some stock header statements output for `#include` files, then the `gen()` methods are invoked on the three

global symbol tables. It finally remains only to copy the scratch file to output, and tack on the main program, which by default starts execution of the compiled system at the agent named SYS.

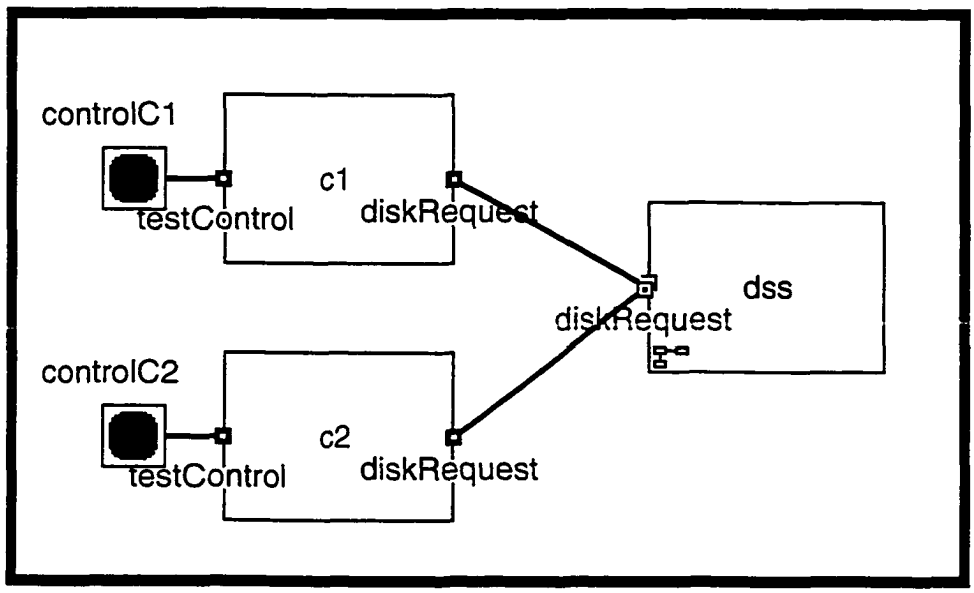
APPENDIX E

Disk Server Modeled in ObjecTime

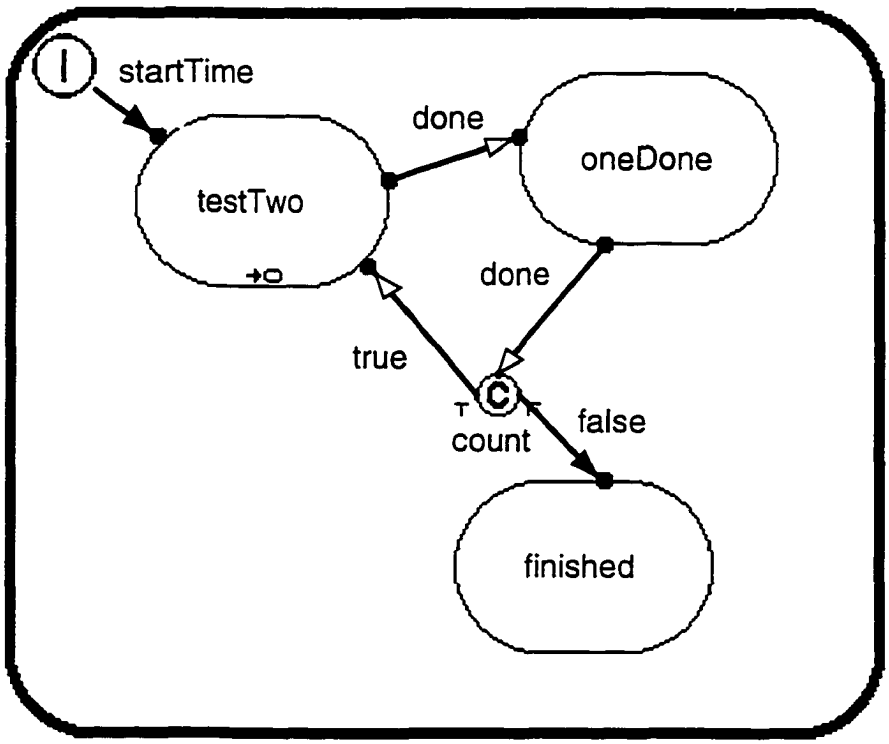
The diagrams on the following pages were produced by ObjecTime Developer 5.2.1. They show the structure of the disk server system in terms of *Actors* (rectangles), the basic structural components in ObjecTime. Actors are similar to CSP processes. They have communication ports (squares) through which they send and receive message signals. Port bindings are shown by lines, and are similar to CSP channels.

The behaviour of each Actor is specified by a finite state machine (FSM), which is also shown below. In FSM diagrams, the labels on the transition arrows are merely textual annotations. The actual triggering events and C++ action code are not printed in these diagrams.

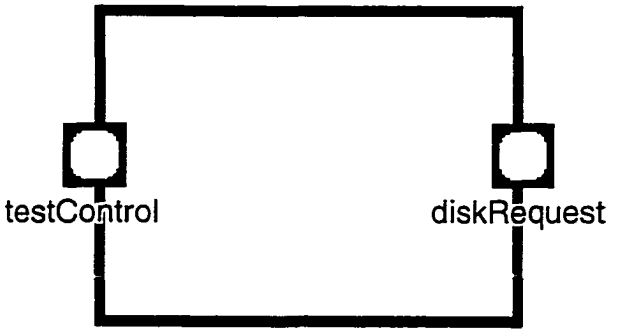
Both Actors and FSMs (actually, ROOMcharts) can be hierarchically decomposed into subcomponents. Actors that are purely structural do not have an FSM specified.



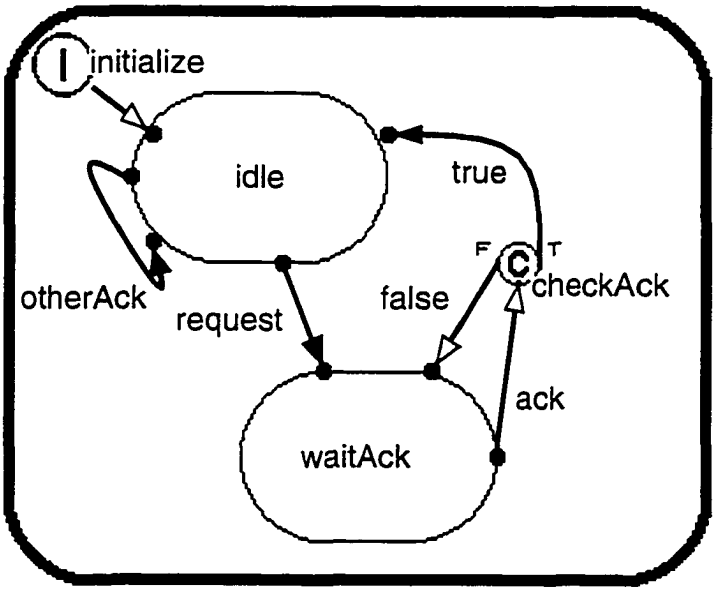
-- ACTOR CLASS: Sys



ACTOR CLASS: Sys STATE: top

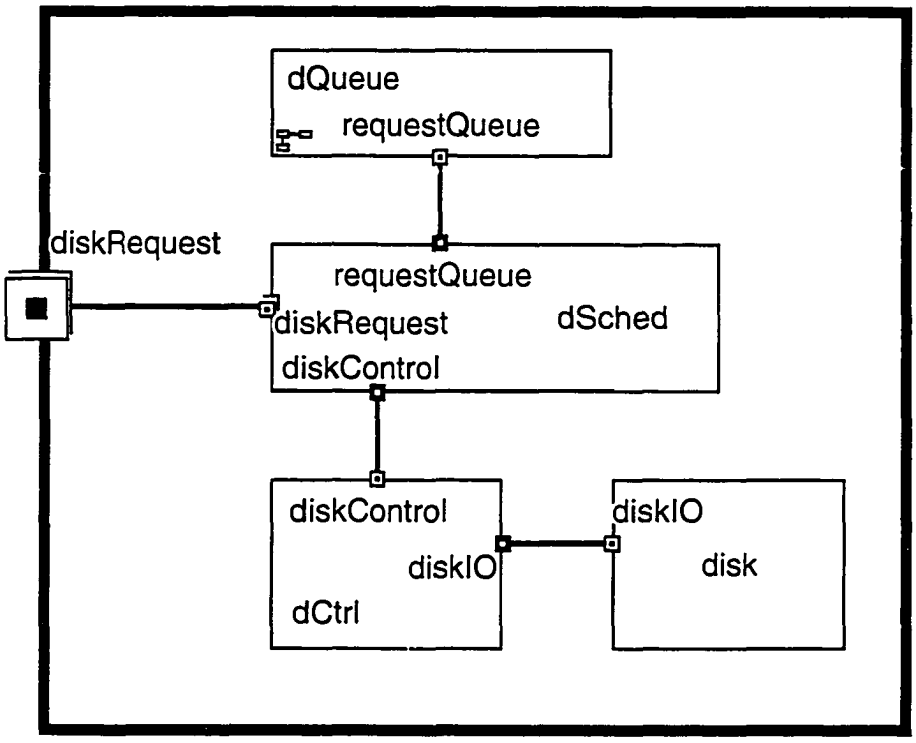


-- ACTOR CLASS: C1

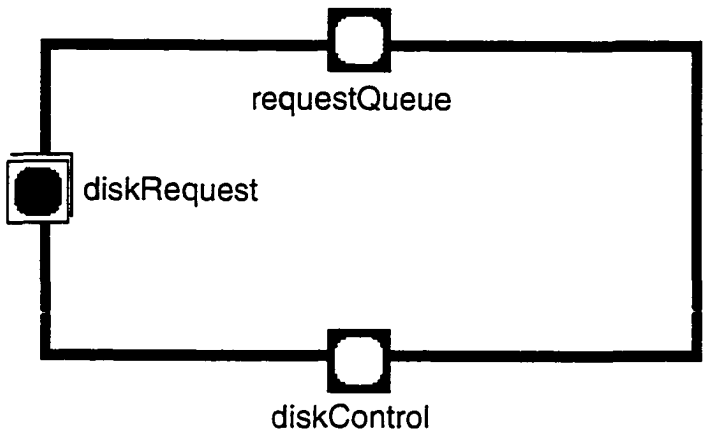


ACTOR CLASS: C1 STATE: top

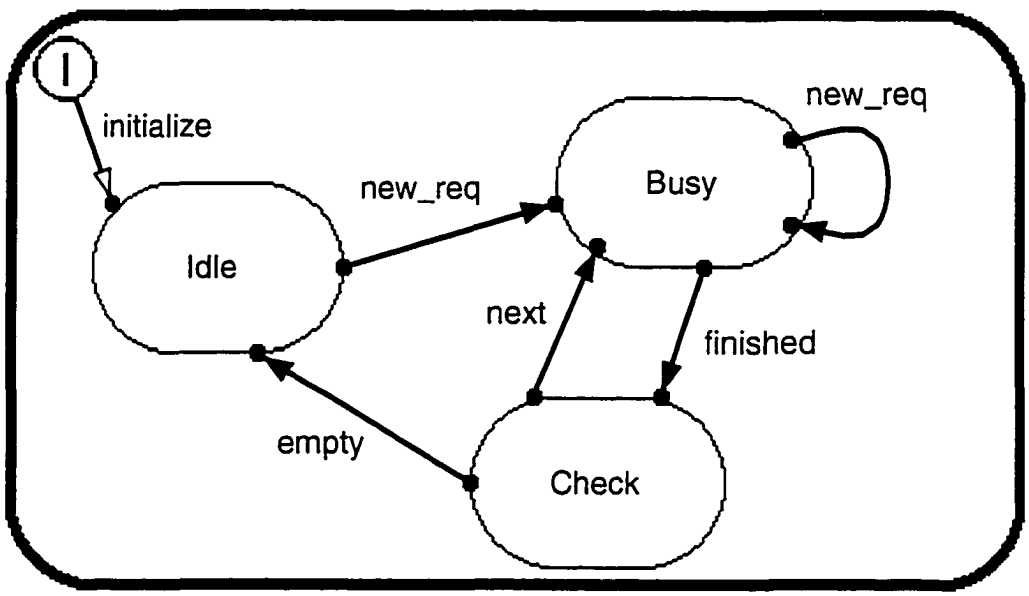
ACTOR CLASS: C2 (same as C1)



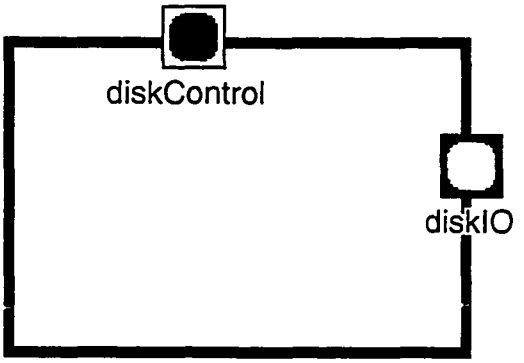
-- ACTOR CLASS: Dss



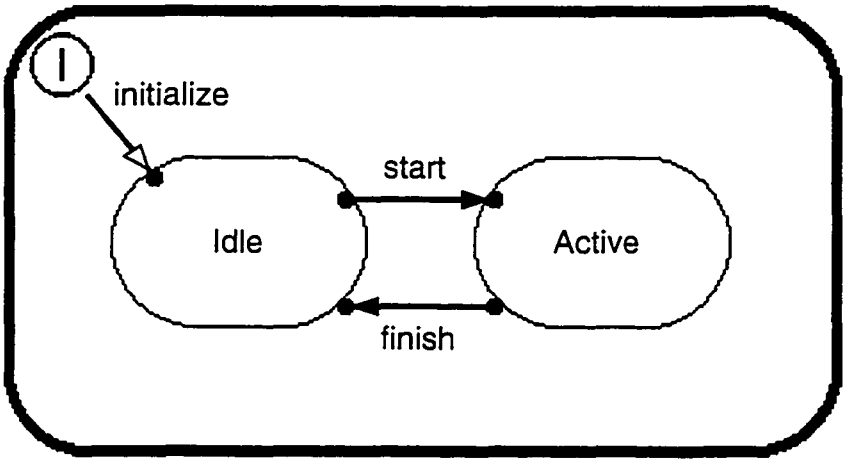
-- ACTOR CLASS: DSched



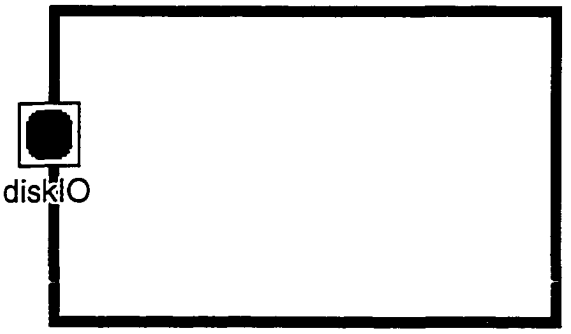
ACTOR CLASS: DSched STATE: top



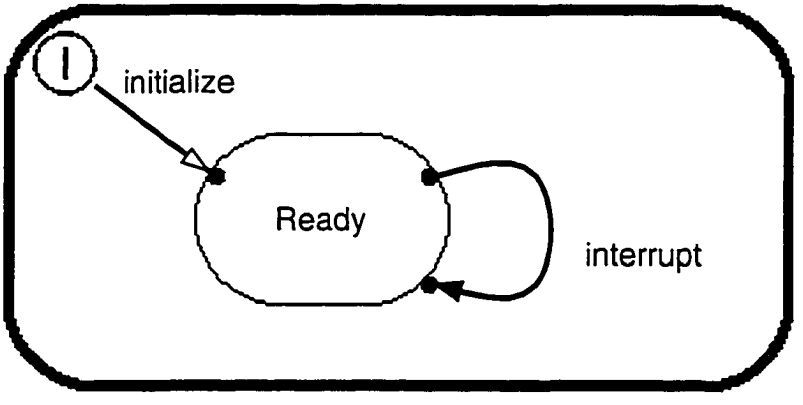
-- ACTOR CLASS: DCtrl



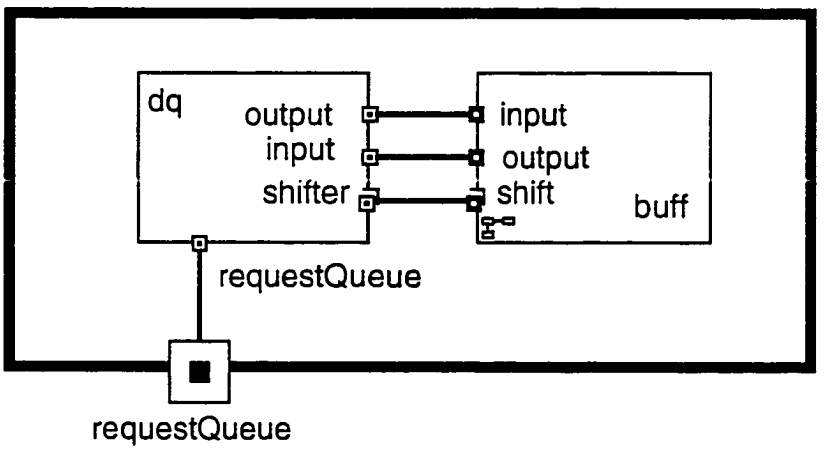
ACTOR CLASS: DCtrl STATE: top



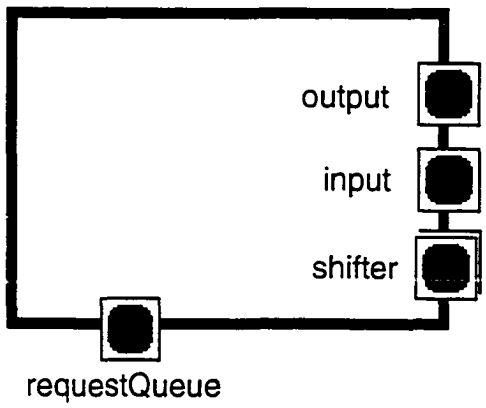
-- ACTOR CLASS: Disk



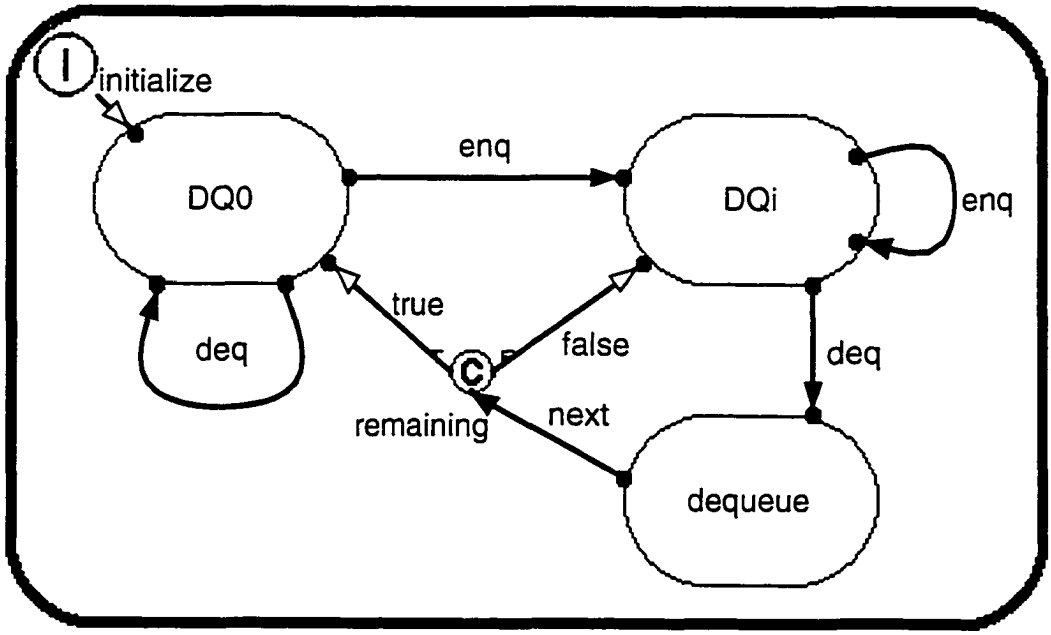
ACTOR CLASS: Disk STATE: top



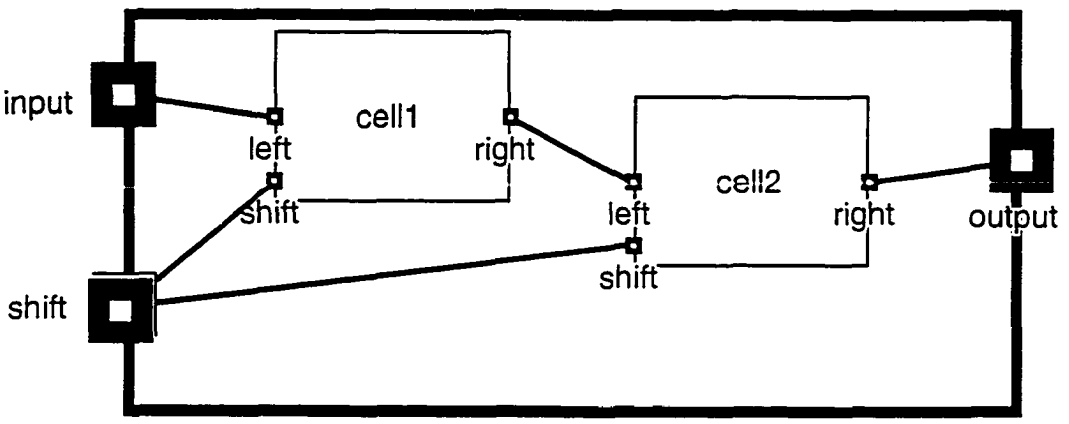
-- ACTOR CLASS: DQueue



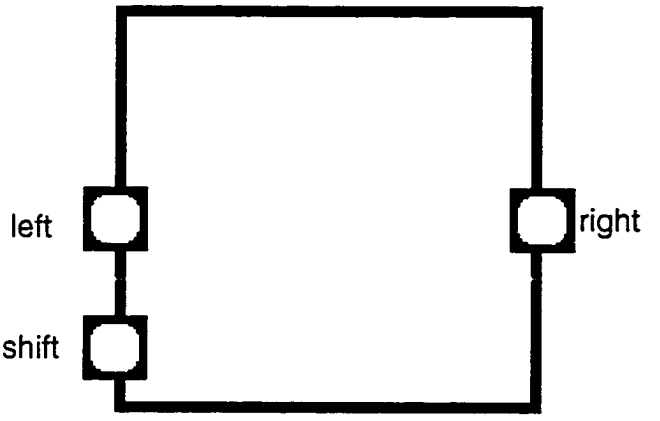
-- ACTOR CLASS: Dq



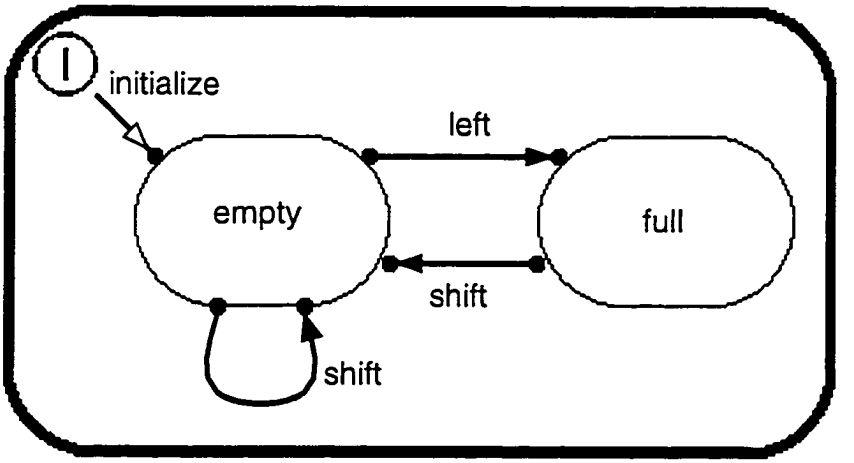
ACTOR CLASS: Dq STATE: top



-- ACTOR CLASS: Buff



-- ACTOR CLASS: Cell



ACTOR CLASS: Cell STATE: top

Glossary

The glossary is divided into sections for Acronyms and Technical Terms. Note: *italics* indicate terms that appear elsewhere in the glossary.

Acronyms

ACP	Algebra of Communicating Processes
BNF	Backus Naur Form
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes, a design formalism that uses algebraic statements to model a system in terms of concurrent *processes*.
FDR	Failures–Divergence Refinement, a software tool based on *CSP* which allows the automatic checking of many properties of finite state systems and the interactive investigation of *processes* which fail these checks.
HLL	High-level language
LOPNP	List of `ParseNode` pointers
OO	Object-oriented
OOAF	Object-oriented application framework
OS	Operating system
POSIX	Portable Operating Systems Interface
STL	C++ Standard Template Library
UML	Unified Modeling Language

VHDL Literally, "VHSIC Hardware Description Language," where VHSIC in turn stands for "Very High Speed Integrated Circuit."

VLSI Very large scale integration (=integrated circuit)

Technical Terms

Action A CSP++ class that corresponds to a *CSP event*. Actions can be used for synchronizing with other *agents* or interfacing with user-coded external routines. Subclasses are "atomic" actions, which do not pass data, and "channel" actions, which do (see *channel*).

Agent A CSP++ class that implements a *CSP process*. In software it would constitute a schedulable thread of control.

Channel (*CSP*) A named unidirectional, nonbuffered interprocess communication port. When two *processes* engage in an *event* whose name is a channel, data is passed from the process using the output symbol (!) to the process using the input symbol (?).

Composition, parallel vs. interleaved (*CSP*)
Designating that a set of *processes* is to execute concurrently. Parallel composition allows for communication and synchronization of the composed processes; interleaved composition does not.

Event (*CSP*) An abstract named activity that a *process* engages in. Events are often defined to represent real-world occurrences originating in the system or its environment.

Executable specification
A high level description of a system that, in addition to its descriptive use, also functions as source code for simulation, logical verification, and/or *synthesis*. *VHDL* is an executable specification language.

Hardware — As the term is used here it primarily refers to digital logic, typically built in the form of integrated circuits or by configuring field-programmable devices.

Process (*CSP*) — An abstraction for a locus of control that engages in a sequence of *events*, some of which may synchronize it with other concurrent processes. A process may be defined in terms other processes.

Synthesis, hardware and software — The automated processing of a specification into a hardware or software end product. In the case of software, the end product is binary machine code, or source code that can be readily compiled into it. In the case of *hardware*, it means a manufacturable circuit description (e.g., netlist), or source code (such as *VHDL*) from which the circuit description can be automatically created.