Using Multiple Representations for Efficient Communication of Abstract Values

by

Earl DeWitt Waldin III

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1992

© Massachusetts Institute of Technology 1992. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
September 4, 1992

Certified by .....
William E. Weihl
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Campbell L. Searle
Chair, Departmental Committee on Graduate Students

ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 30 1992
LIBRARIES

Using Multiple Representations for Efficient Communication of Abstract Values

by

Earl DeWitt Waldin III

Submitted to the Department of Electrical Engineering and Computer Science
on September 4, 1992, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Programs in a heterogeneous environment communicate by exchanging typed values in messages. When sending and receiving values, a program must translate between the internal representation used in the program and the transfer representation used in a message. Reducing the overhead of translation is a major factor in achieving efficient communication.

The importance of using efficient message representations has long been recogized for machine-dependent types such as integers and floating point numbers. The solution is to define multiple transfer representations for a given type and allow programs to negotiate the representation actually used. However, existing systems do not allow multiple representations and negotiation for user-defined types. As shown in this dissertation, negotiating an efficient representation for user-defined types is just as important as for machine-dependent types.

The major focus of this dissertation is on choosing the transfer representation that has the lowest translation cost. As shown in this dissertation, quantitative cost models for user-defined types are difficult to describe. In contrast, programmers are able to state qualitative preferences among representations that accurately reflect costs. I present a novel language for expressing qualitative preferences and show its application to two relatively complex types: a video display image and a VLSI mask. I also present efficient algorithms for computing the most preferred representation given two sets of preferences. The qualitative model presented here provides many of the benefits of a quantitative model, but is simpler and easier to use.

To show that negotiation with preferences is practical, I describe a prototype implementation of the algorithms and give measurements. I also show how to incorporate negotiation into existing systems.

Thesis Supervisor: William E. Weihl
Title: Associate Professor of Computer Science and Engineering

This research was supported by the National Science Foundation under Grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, by a grant from AT&T, and by an equipment grant from Digital Equipment Corporation.

Acknowledgments

I would like to thank my advisor, Bill Weihl, for his technical guidance, encouragement, editorial advice, and patience in seeing this thesis through to completion. David Tennenhouse provided invaluable insight on numerous occasions and always found the right words to keep me going when I needed it. I would also like to thank Barbara Liskov for her comments and suggestions.

Anthony Joseph supplied a sounding board for my ideas and proofread various drafts. Wilson Hsieh deserves my special thanks for scrutinizing the final draft. Bruce Leban helped with proofreading and gets the credit for naming the language "Amalgam."

Many thanks to all my fellow journeymen for the good times, bad times, and above all, those multifarious and engaging discussions, productive and otherwise: Wilson, Anthony, Suresh Jagannathan, Bruce, Paul Wang, Sanjay Ghemawat, Boaz Ben Zvi, Bob Gruber, Jeff Cohen, Carl Waldspurger, and many, many others. They prove the age-old adage that getting there is half the fun.

I am grateful to Dr. Morton Hammond for taming my allergies long enough for me to finish.

To my wife Dora, I don't know how you managed it; I know it has been difficult at times. I will never forget your patience, understanding, and love. My two children, Alexander and Nicholas, never failed to provide me with plenty of love and joy. I owe them all the time that I have not been able to give them lately.

Finally, I cannot thank my mother and father enough for their steadfast moral and financial support, especially during the last few years. This thesis would not have been possible without them.

to my family.

Contents

1 Introduction **13**

1.1 Motivation 14

1.2 Background 18

1.2.1 The OSI Presentation Layer 18

1.2.2 Mercury 24

1.3 Related Work 26

1.4 Overview of Thesis 27

2 Vspaces, Representations and Negotiable Sets **31**

2.1 Introduction 31

2.1.1 Vspaces 32

2.1.2 Abstract and Transfer Representations 32

2.1.3 Overview 33

2.2 Specifications 34

2.3 The Language Amalgam 35

2.3.1 Built-in Vspaces and Constructors 36

2.3.2 Abstract Vspaces and Constructors 37

2.3.3 Equates 41

2.3.4 Reference Grammar 42

2.4 Vspace Components 43

2.5 Negotiable Sets 44

2.6 The Registry 45

2.7 Summary 47

3 Contexts, Preferences and Negotiation 49

3.1 Contexts . . . 50

3.1.1 The Function *Components* . . . 51

3.1.2 Assignments . . . 53

3.1.3 Definition of a Context . . . 55

3.2 Preferences and Context Lists . . . 56

3.2.1 Problems with Quantitative Measures . . . 57

3.2.2 Context Lists . . . 61

3.3 Preferences and Negotiation . . . 63

4 Expressing Context Lists 65

4.1 The Approach . . . 65

4.2 More on Assignments . . . 67

4.3 Preference Lists . . . 68

4.3.1 Preference-List Operators . . . 69

4.3.2 Union Operators . . . 70

4.3.3 Join Operators . . . 73

5 Abstract Vspace Examples 79

5.1 Video Display Image . . . 79

5.1.1 The Color Model . . . 81

5.1.2 Organization of Components . . . 84

5.1.3 Mapping Concepts Into Vspace Descriptions . . . 86

5.1.4 Preference List . . . 88

5.2 VLSI Design . . . 90

5.2.1 The Abstraction . . . 91

5.2.2 Representations . . . 93

5.2.3 Preference List . . . 104

6 Implementing Context Lists 111

6.1 Some Notation . . . 113

6.2 Computing the Best Context . . . 113

6.2.1 Adjacency-List Representation . . . 114

6.2.2 Equivalence-Class List Representation . . . 118
6.3 Representing Contexts . . . 123
6.4 The Preference-List Abstraction . . . 124
6.4.1 Conjunction . . . 126
6.4.2 Dominant Union . . . 126
6.4.3 Consistent Join . . . 127
6.4.4 Dominant Join . . . 134
6.5 Implementation of Assignments . . . 134
6.6 Overview of the Prototype Implementation . . . 138
6.7 Measurements . . . 141
6.7.1 The Test Bed . . . 141
6.7.2 The Example Context Lists . . . 142
6.7.3 Test Cases and Data . . . 143
6.7.4 Analysis of Results . . . 145

7 Negotiation in Real Systems **151**
7.1 Negotiation in Mercury . . . 151
7.1.1 Mercury Ports and Interfaces . . . 152
7.1.2 Negotiable Sets . . . 154
7.1.3 The Binding Model . . . 156
7.1.4 Specifying Context Lists . . . 158
7.1.5 When to Negotiate . . . 160
7.1.6 What to Negotiate . . . 161
7.1.7 Using CIDs . . . 162
7.1.8 Summary of Design Tradeoffs . . . 163
7.2 Negotiation in OSI . . . 164
7.2.1 OSI in General . . . 164
7.2.2 Negotiating for ROS . . . 169
7.3 Negotiation in OSF/DCE . . . 170

8 Veneer Design for Mercury **173**
8.1 Negotiation Subsystem . . . 175
8.2 XDR subsystem . . . 182

8.3 Client Stub 184
8.4 Server Stub 188
8.5 Summary 190

9 Conclusion **195**
9.1 Summary 195
9.2 Analysis of Design Decisions 197
9.3 Suggestions for Future Work 202

A Matrix Marshalling Code **205**

B Abstract Vspace Conversion Issues **213**

Chapter 1

Introduction

In recent years the number of computers tied together through both local and wide-area networks has increased dramatically. With the introduction of networks running at one hundred megabits per second, and with gigabit data rates around the corner, the amount of information exchanged is rising dramatically. In addition, as computers have become more powerful, the kind of data exchanged has become qualitatively more sophisticated. Already there are design environments where computer-aided design information on everything from VLSI chips to automobiles is communicated. The introduction of multimedia and computer-mediated cooperative work environments can only accelerate this trend.

What all the networks and applications mentioned above have in common is that they typically operate in a heterogeneous environment, where application programs that communicate are written in different programming languages and run on computers made by different manufacturers. In such an environment, communicating programs often represent the same data differently. In such cases, data must be translated from one representation into another. A judicious choice for the representation used to communicate data is required in order to minimize translation cost.

In this thesis, I present and analyze methods that allow programs to negotiate efficient representations for communicating values of any data type, including user-defined types. The methods are based on the use of abstraction for defining data types, a system for naming types and representations, a novel language for specifying

preferences among representations, and a set of algorithms for choosing the lowest-cost representation given a set of preferences from two programs.

Because existing systems have not recognized the need to negotiate representations for user-defined types, I include below an extensive section on motivation that demonstrates its importance. This is followed by background material that lays the groundwork for later material and exposes flaws in existing systems that prevent negotiation for user-defined types. The chapter concludes with a brief overview of the remainder of the thesis and the expected contributions.

1.1 Motivation

The importance of choosing "efficient" transmission representations has long been recognized for machine-dependent scalar data types such as integers [Jam90, Coh81, HD89]. Given that the bulk of transmitted data consists of scalars, much of the effort in designing transmission representations has focused on them. Scalars are also relatively expensive to convert. For example, converting 32-bit integers from "big-endian" byte order to "little-endian" order requires swapping the bytes, which can take from 5 to 10 CPU cycles on current RISC processors.

A number of systems choose a canonical representation for sending data [Sun87, Xer81, HL82, JRT85]. Because the representation is known in advance, it is not necessary to embed information in the data stream that describes the data's format. In addition, the transmission representation for a type is typically chosen so that it closely matches the internal representation used by most systems. For example, a 32-bit integer is represented using four bytes that are aligned on a four-byte boundary. The result is that marshalling is usually efficient. In the case where the canonical representation is identical to the internal representation, large data structures can be marshalled using a block-move operation, which is very efficient.

However, canonical representations can often require *double translation*, which occurs when two systems that use the same internal representation must translate data to and from a different canonical representation, once on transmission and once on reception. No matter which representation is chosen as the canonical one, there will always be systems that differ.

A few systems allow applications to choose the transmission representation at runtime [DLM+87, Jos88, ISO84]. The best known and most general system for negotiation is found in the presentation layer of the ISO Open Systems Interconnection (OSI) standards [ISO88d]. As I discuss later, negotiation in OSI is currently limited to built-in types such as scalars, sequences and sets.

Given that even the most structured data consists primarily of scalars, it would appear at first glance that canonical representations for user-defined types are not a problem. However, as the following two examples show, the cost of translation for user-defined data types is comparable to, and sometimes even greater than, the cost of translating scalars.

For the first example, consider a two-dimensional matrix of 32-bit integers. In the programming language C, the matrix elements are stored sequentially in memory in row-major order, while in Fortran, they are stored in column-major order. If a matrix is transmitted between two C programs, or between two Fortran programs, then there is no need to transpose the matrix. In contrast, if communication is between a C and a Fortran program, then the matrix must be transposed. If we include the possibility of different machine architectures for integers, then there are four combinations of byte-swapping and transposition that can occur.

The different marshalling costs for the four combinations are illustrated by the graph in Figure 1-1, which gives measurements of the cost of marshalling each combination for transmission on a DECstation 5000/200.[1] The numbers above each bar in the graph give the average number of CPU cycles required to marshal one matrix element and include loop overhead, as well as the overhead for breaking the matrix into packets of 1,024 bytes in length. The cost of allocating and deallocating packets is not included.[2] The bar labeled "block move" corresponds to the case where both programs use the exact same representation, in which case the elements are placed in the packet buffer using a very efficient block move loop. If the hardware and operating system supported direct transmission of program data without the need for copying, then the cost in this case would be close to zero. The labels on the

[1] DECstation is a trademark of the Digital Equipment Corporation.

[2] The measurements also do not include processor cache effects, which can be an important factor under certain circumstances.

other bars are self-explanatory. A more complete description of the program code and measurement techniques is given in Appendix A.

As shown in the graph, transposing the matrix costs about seven cycles/integer more than a block move. This may not sound like much, but even a 10×10 matrix costs an additional 700 cycles. In comparison, 700 cycles is approximately the cost of sending a call message on systems that have optimized the code path for a remote procedure call [SB89, vRvST89]. A larger matrix of 100×100 would incur an additional 70,000 cycles, which is a significant overhead. In the case of double translation, the overhead occurs at both the sender and receiver, which increases latency even more.

In practice, the costs of transposing would be higher than those given in the graph because the transposition code in Appendix A was highly optimized by hand to integrate transposition with buffering and packetizing. This is unlikely to be the case when marshalling code is generated by a "standard" stub compiler. Unless the application programmer has access to the low-level interface used by the compiled stub, the application program must transpose the matrix before presenting it to the stub for marshalling and packetizing. In this case, the cost of transposing could be higher than that for byte swapping because two passes are required, one for transposing, and one for buffering and packetizing.[3] The cost of byte swapping, on the other hand, remains the same because it is supported at a low level by the stub compiler.

As a second example, consider a VLSI design environment with a a number of CAD tools for designing, manipulating and verifying VLSI masks. A VLSI mask consists of a hierarchy of cells, where a cell at one level contains one or more cells from a lower level. A cell at the lowest level is a single geometric figure, such as a rectangle, that belongs to one physical layer on the chip, e.g., polysilicon or diffusion. In addition, a cell at one level can be shared, transformed and replicated by enclosing cells at a higher level.

There are many tools that interact with masks in a VLSI design environment.

[3]This case provides an example of when cache effects can become important. If the matrix is large enough, then the output of the first pass may not be in the cache when it is accessed during the second pass.

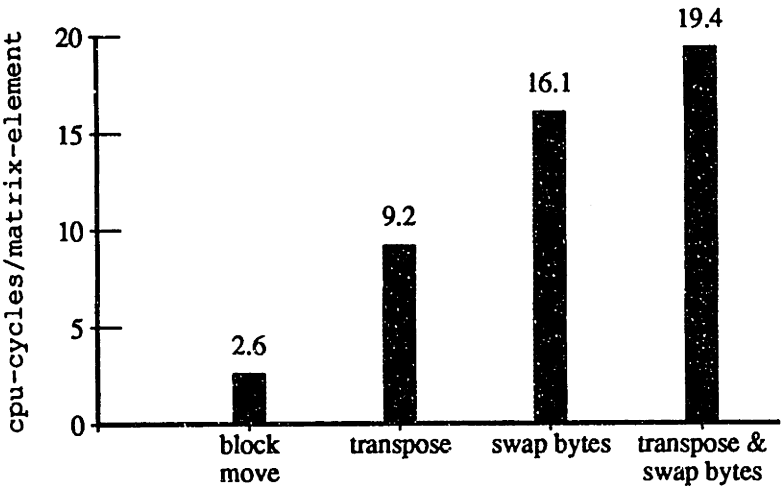


Figure 1-1: Marshalling costs for 2D matrix on a DECstation 5000/200

One well known canonical representation for communicating masks between tools is the California Intermediate Form (CIF), which is a human-readable language for describing cells [MC80]. Similar representation languages have been created by manufacturers of chip processing equipment and are used in industry [Mur90]. However, most tools use an internal representation with more structure, e.g., quad trees [Sam88, Ked82, NR86], corner stitching [Ous84, Ous85], linked arrays [HA90], and bit rasters [BT80]. If the transmission representation is the same as the internal one, encoding and decoding are linear in the size of the data and no additional translation is necessary.

In the case where a tool's internal representation differs from the transmission representation, communicating a VLSI mask description requires translation. Moreover, translation can be costly, as I show in Chapter 5, where this example is treated in more detail. For example, if the internal representation is corner stitching and the transmission representation is CIF, then the conversion process for transmission requires many tens of instructions, if not a few hundred, to translate each cell for transmission. For reception the situation is even worse; translation requires $O(n(R^{1.5} + S^{1.5}))$ time, where n is the number of cells, R is the average number of rectangles in a cell, and S is the average number of subcells in a cell. Because masks contain many thousands of cells, the additional overhead of translating can be substantial.

The VLSI mask example also illustrates the difficulty of choosing the most efficient transmission representation in the case where two applications use different internal representations. A VLSI mask is built from many layers of abstraction, e.g., a mask consists of "cells" that contain "rectangles" that are built from "points" that are specified by "coordinates." Each of these abstractions can have more than one representation, leading to many possible combinations. The problem then is how to specify and select the combination that minimizes the overhead of translation.

In summary, the translation overhead for user-defined types is as important as that for low-level types, and sometimes even more so as demonstrated by the non-linear factors in the VLSI mask example above. What is needed are mechanisms that allow applications to negotiate representations for user-defined types, thereby avoiding double translation when possible. In addition, negotiation should choose the most efficient representation when programs use different internal representations.

1.2 Background

The work in this thesis extends the work found both in the presentation layer of the ISO standards for Open Systems Interconnection (OSI) [ISO88d] and in Mercury, which is a communication system for heterogeneous programming environments that is based on the remote procedure call (RPC) paradigm [LBG+88]. The OSI presentation standard provides a general framework for negotiating transmission representations at runtime. However, as shown below in Section 1.2.1.2, its naming mechanisms are insufficient for negotiating representations of user-defined types. Extending the naming mechanisms requires changing the presentation model defined in [ISO88d]. Mercury, on the other hand, provides mechanisms for naming user-defined types that can be extended to allow negotiation. These systems are described in the following two sections.

1.2.1 The OSI Presentation Layer

This section reviews the design of OSI's presentation and negotiation model and exposes problems that prevent the definition and negotiation of multiple represen-

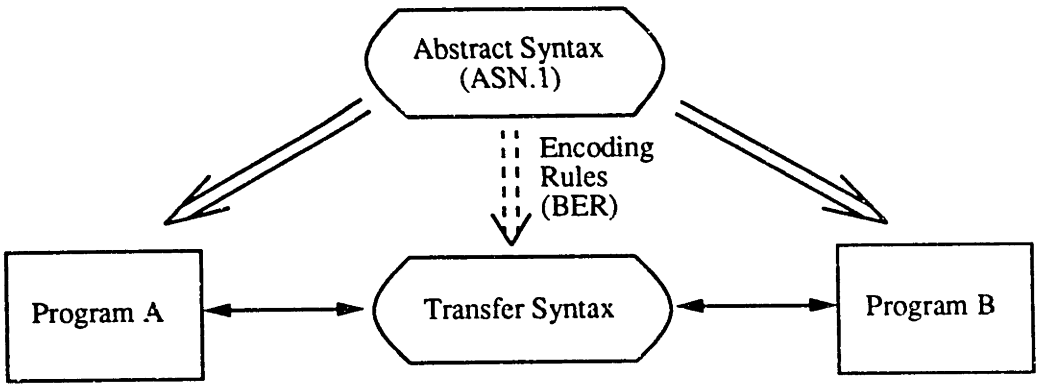


Figure 1-2: Model of the OSI presentation layer

tations for user-defined types. The description of the model is somewhat extensive because it describes not only a general framework, but also a specific implementation. Familiarity with the OSI presentation layer will also help later with understanding the model introduced in this thesis.

1.2.1.1 The Model

The OSI presentation layer uses a two-level approach that separates the definition of a type (i.e., the set of abstract values that constitutes the type) from the encoding used to transmit the type's values. A type is defined by an abstract syntax and its encoding by a transfer syntax. The division into abstract and transfer syntaxes is illustrated in Figure 1-2. The arrows between the boxes labeled "Program A," "Transfer syntax," and "Program B" indicate the communication path. The double arrows originating from the box labeled "Abstract syntax" are described later. The discussion below begins with a description of abstract and transfer syntaxes, and then shows how they work together to allow different systems to transmit the same values using different encodings.

An *abstract syntax* is a specification of application-layer data or application protocol/control information. In essence, an abstract syntax specifies the interface for a given application by defining the types and values that can appear in messages sent and received by that application. The types and values are specified using an *abstract syntax notation.* A given abstract syntax can be unambiguously identified by an *abstract syntax name*, which is a unique object identifier assigned in

accordance with an ISO standardization process [ISO87a, annexes B, C, and D]. An abstract syntax name, therefore, uniquely identifies a specific collection of types and distinguished values.

ASN.1 (Abstract Syntax Notation one), is an example of a notation for defining abstract syntaxes [ISO87a]. It is currently used to define the interface for both standard application components, such as the remote operation facility [ISO88b, ISO88c], and standard applications, such as the OSI directory [ISO88a]. ASN.1 provides a grammar for defining both types and distinguished values of those types. It is analogous to the type declaration portion of a high-level programming language like Pascal or C..

The ASN.1 specification defines a number of built-in types, such as "integer," "real" and "boolean." It also defines several type constructors, such as "set" and "sequence," that allow the specification of structured data types. A special type constructor, called *tagged type*, is used to specify a named user-defined type. A tagged type is specified by referencing another type and giving it a new name called a *tag*; the newly named type is isomorphic to the other type, but is considered distinct from it.

A *transfer syntax* specifies a concrete (i.e., bit-level) representation for a set of types. The concrete representation for a given type determines how the values of that type are encoded for transmission. A transfer syntax is defined for a specific abstract syntax.

One (very tedious) way to define a transfer syntax is to give the encoding for each type in the abstract syntax explicitly. A more common way is to use a set of rules that generate an encoding for any type described by a specific abstract syntax notation. For example, there is a standard set of *basic encoding rules* (BER) for use with ASN.1 [ISO87b]. Thus, given an abstract syntax specified by some notation, we can automatically generate a transfer syntax by applying the rules. This is illustrated in Figure 1-2 by the double arrow marked "(BER)."

A *transfer syntax name* is a name that unambiguously identifies either a transfer syntax or a set of rules for generating a transfer syntax for a given abstract syntax. As with abstract syntaxes, transfer syntaxes are assigned unique names via an ISO standardization process.

Abstract and transfer syntaxes work together in pairs. An association of an abstract syntax with a transfer syntax is called a *presentation context.* Given a presentation context one can encode and decode every value of every type defined in the abstract syntax. A presentation context is uniquely identified by a pair of abstract and transfer syntax names.

The division of function into abstract and transfer syntaxes allows the presentation layer to use different transfer syntaxes and, therefore, different encodings, for the same abstract syntax. The presentation layer has a negotiation mechanism whereby two presentation-layer peers can agree on the transfer syntax to use for a given abstract syntax. The application layer does not participate in the negotiation.

Negotiation works as follows. One peer, the originator, sends an abstract syntax name together with a list of transfer syntax names that it can handle for that abstract syntax. The other peer then selects one transfer syntax that it can handle from the list and communicates its choice to the originating peer. The resulting abstract-transfer syntax pair then becomes the presentation context for transmission of data. The negotiation does not allow the peers to negotiate an efficient choice (as is a goal of this thesis) — only a workable one.

The primary advantage of using ASN.1 to specify an abstract syntax is that it allows application programmers to specify a type's representation at a high level of abstraction, a level similar to that found in programming languages. Moreover, this similarity makes it possible to provide tools that automate the process of writing the marshalling routines that translate between the internal representation of a program and the transfer syntax [CM88, WSG91, Gib87, HL82]. This is illustrated in Figure 1-2 by the double arrows from "Abstract syntax" to "Program A" and "Program B." Given a specification in ASN.1, one can automatically generate the interface used by a program to communicate with other programs.

The success of ASN.1 as a tool for specifying international standards is due to the fact that one can write a single specification in ASN.1 that determines indirectly both a transfer syntax and a program interface. Individuals writing standards can focus on specifying only ASN.1 type definitions, knowing that there are programming tools that take care of the remaining details of communication.

1.2.1.2 Problems with OSI

The fact that ASN.1 indirectly defines a transfer syntax is also its biggest problem with respect to negotiating representations for user-defined types. The use of ASN.1 together with any set of encoding rules means that an ASN.1 type definition specifies not only a set of values, but also a representation. Consider the following two definitions for a rectangle written in ASN.1:

```
Point ::= [0] SEQUENCE {x INTEGER, y INTEGER}
Rectangle ::= [1] SEQUENCE {height INTEGER,  width INTEGER,
                            lower-left-corner Point}
```

and

```
Rectangle ::= [2] SEQUENCE {x-min INTEGER,  x-max INTEGER,
                            y-min INTEGER,  y-max INTEGER}
```

where $[n]$ denotes a tag for the newly defined type and a SEQUENCE is similar to a record type in a programming language. The first definition for `Rectangle` represents a value by giving its height, width, and the location of the lower left-hand corner. The second represents a value by giving the maximum and minimum values of the x and y coordinates of the rectangle's four corners.

It is clear that both definitions define the same abstraction in that we can convert any value of either rectangle type to a value of the other. However, each definition specifies a different way of representing the bounds and location of a rectangle. Furthermore, a given abstract syntax specification must either choose one of the two representations or combine them into a CHOICE data type.[4] Choosing only one representation corresponds to choosing a canonical representation, with the consequence of double translation.

The following definition shows how a CHOICE data type combines the two definitions for rectangle:

[4] A CHOICE data type is similar to a disjoint union in most programming languages.

```
Rectangle ::=
   CHOICE {lower-left [1] SEQUENCE {height INTEGER, width INTEGER,
                                    lower-left-corner Point}
           bounded    [2] SEQUENCE {x-min INTEGER, x-max INTEGER,
                                    y-min INTEGER, y-max INTEGER}}
```

The tags "1" and "2" above are what is important; the purpose of the identifiers `lower-left` and `bounded` is not relevant here. Any transfer syntax for the above CHOICE must include the tag as part of the encoded value. Otherwise, the receiver could not determine which representation was used to encode the value.

With a CHOICE data type, any application is free to send rectangles using either representation. This has the same effect as using the negotiation strategy known as "receiver makes it right." In this strategy, the sending program chooses the representation that best matches its internal representation. The receiving program is responsible for translating between representations, if necessary. This strategy avoids double translation if the representation chosen by the transmitter matches the internal representation of both the transmitter and the receiver. However, it suffers from the following problems:

1. It requires every receiver to handle every representation.

2. Evolving the definition to include new representations or delete existing ones requires additional mechanisms to insure that the receiver can always decode a value.

3. It does not provide a mechanism for negotiating the most "efficient" representation in the case where the peers use different internal representations. The choice is made unilaterally by the sender.

Given the existing presentation-layer framework, the only alternative to using a CHOICE is to define the representation of a rectangle within a transfer syntax. However, it is not desirable to do so because a transfer syntax is too low-level. One loses the benefit of using an abstract syntax notation to generate the interface and marshalling routines used by a program.

In spite of its shortcomings, the CHOICE type suggests an approach to providing negotiation for user-defined types. If, during negotiation, each peer tells the other

which tags it can interpret for each user-defined type, then problems 1 and 2 above can be eliminated.[5] Problem 3 cannot be solved so easily. In fact, it is one of the main topics of this thesis.

One drawback of the above approach is that a tag must be included in the encoding of every value of a user-defined type in a message. When a type consists of several layers of user-defined types, then there is a tag for each layer of each value. The presence of these tags can introduce substantial translation overhead, especially in the case where the type is a collection of values, e.g., a set of rectangles. The approach used in this thesis is to remove the tags from the encoded values and incorporate them into the presentation context. Given the presentation context used to encode a message, the receiver can then decode all the values in that message.

Recall from Section 1.1 that one of the advantages of a canonical transfer representation is that there is no need to embed information like tags in the encoded data stream. If the transfer representation matches a program's internal representation, then translation can be made efficient by using block-move operations or direct memory access for I/O devices. By moving the information provided by the tags out of the data stream and into the presentation context, we recover the advantages of using a canonical representation while retaining the advantages of negotiation.

1.2.2 Mercury

Mercury is an RPC-based system for building distributed systems in a heterogeneous environment [LBG+88]. A remote procedure in Mercury is called a *port*. As in other RPC-based systems, ports can be created, exported, and invoked. One aspect of Mercury that distinguishes it from other systems is that ports are first-class objects; a port can be passed as an argument or result in another port invocation.

Ports have typed arguments and results that are described using an interface definition language (IDL). Several ports can be grouped into an *interface*, which provides an "object-like" definition of a service. In addition, the IDL allows programmers to create new user-defined types and type constructors. The definitions of

[5]This assumes there is at least one designated representation that all programs can use so that communication is always possible.

types, ports and interfaces all reside in the Mercury *registry*, which is a distributed repository.[6]

One important goal of Mercury is to provide support for data abstraction. With respect to this thesis, the most relevant parts of Mercury are its mechanisms for naming abstract types and type checking their use. The naming mechanism has its origins in Argus, which is an experimental programming language/system that supports the construction and execution of distributed programs [Lis88, Lis87b]. Argus, in turn, is an extension of CLU, a programming language that was also designed to support abstract data types [LG86].

Every type in Mercury is assigned a unique name [LW86]. For built-in and abstract types, the name is a unique identifier (UID) that is assigned by the Mercury registry. The registry also assigns a UID to every built-in and abstract type constructor. The name of a constructed type is a tree-like structure formed from the constructor's UID and the names of the component types. For the purpose of type checking, two types are the same if they have the same name. For two constructed types the names must have the same structure. In programming language terms, built-in and abstract types use name equivalence and constructed types use structural equivalence.

The most important point about Mercury's type checking is that a program's use of a port or interface can be type checked *independently of the representation used for transmitting values of any type.* The UIDs act as surrogates for the representation. Type checking interfaces at this level of abstraction allows for the possibility of late binding for the transmission representation. Because the UIDs assigned by the registry have a unique interpretation, they can be used to negotiate a representation at runtime, assuming there is also a mechanism for naming representations.

Binding representations at runtime is non-trivial, and the early research on Mercury focused more on the communication mechanisms and protocols. As a result, the initial design allowed only one representation for each data type. However, its framework for naming provides a base upon which to add negotiation of representations for any type, including user-defined types.

[6]The interface description language and the structure of the registry are discussed in the following design notes: [LW86, Wei87a, Wei87b, Wei88].

1.3 Related Work

In addition to OSI and Mercury, which are discussed above, there are a number of existing systems that address the problem of communication in heterogeneous systems: Courier [Xer81], Sun-RPC [Sun85, Sun87], Matchmaker [JRT85, JR86], OSF/DCE [FS91], Apollo [DLM+87], MLP [HS87], Argus [Lis88], HRPC [BCL+87], and Alliance [Jos88]. Of these systems, Sun-RPC, MLP, and Argus use canonical representations for both built-in and user-defined types. As discussed earlier in Section 1.1, canonical representations can result in double translation. Matchmaker, Apollo, and OSF/DCE, which is based on Apollo, allow a small but fixed number of representations for machine-dependent scalars and use the strategy of "receiver makes it right" described earlier on page 23. This avoids the problem of double translation, but suffers from other problems already mentioned. HRPC is the RPC facility created as part of the Heterogeneous Computer Systems (HCS) project at the University of Washington [NBL+88]. HRPC accommodates system heterogeneity by emulating a wide variety of existing RPC facilities. The primary goal of HCS is to reduce the cost of integrating a new system type into a computing environment. Although HRPC provides a number of good ideas for structuring the implementation of a client or server, it does not address the goals of this thesis. This leaves Courier and Alliance, which I discuss in the remainder of this section.

Courier is an RPC-based system that uses versions to allow change. A given version specifies a canonical representation for all data types. Versions are numbered in order (e.g., 1, 2, ...), and different versions may use different representations. When a client program contacts a server, the client specifies the range of versions it supports. The server picks the highest version from this range that it supports as the version to use. Although originally intended to provide for system evolution, versions could be used to support multiple representations. However, versions are not useful in practice because it is assumed that: 1) there are a small number of versions, 2) the versions can be linearly ordered, and 3) the most recent version is the "best" one to use.

Alliance is an RPC-based system that allows multiple representations for built-in types and type constructors. Each representation is assigned a unique identifier, and

the representation of each type is negotiated separately. Although less well known than other systems, it is interesting because it uses a quantitative cost measure as the basis for negotiating the lowest-cost representation. However, the problem of how one assigns a cost to each representation was left as an open research question. In addition, it is not clear if Alliance's cost model can be extended to deal with user-defined types. For example, the non-linear factor for marshalling VLSI masks cannot be expressed using the current cost model. As shown later in Chapter 3, even if the model could be extended, it is not always possible to assign a quantitative measure.

With the exception of Alliance, none of the above systems provide mechanisms that are as general as those found in the OSI presentation layer. Neither Alliance nor OSI provide mechanisms for user-defined types. This thesis presents mechanisms that not only avoid double translation for user-defined types when possible, but also choose an efficient representation in the case where two programs use different internal representations.

1.4 Overview of Thesis

The main focus of this thesis is the efficient communication of values of any type, with primary emphasis on user-defined types. Although the issues are most apparent in distributed systems where values are transmitted over a network, the problem being addressed is communication in a heterogeneous environment. More specifically, the goal of this thesis is to investigate and design mechanisms for defining transfer representations for user-defined types and for determining the transfer representation that minimizes the cost of translation.

Achieving the above goal requires making several design decisions from a large number of possible choices that can interact in complex ways. One of the guiding principles used to tame complexity during design is simplicity. Keeping things simple makes them easier to understand and use. It also makes it easier to analyze the interactions among different parts of the design.

A secondary goal of this thesis is extending Mercury to provide negotiation of representations for user-defined types. The approach incorporates many of the

concepts present in OSI and expands some of the ideas outlined at the end of Section 1.2.1.2. There are two reasons for using Mercury. The first is that the Mercury environment provides a framework for abstraction and specification, which provides an ideal starting point. The second is that Mercury is an RPC-based system, which restricts the focus to negotiating for application-level types. OSI is more general in that it addresses the additional problem of defining types used in application-level protocols. In spite of restricting the focus to Mercury, there are a number of results that are applicable to OSI. Later chapters include brief descriptions of how these results can be incorporated into OSI.

The first part of the thesis extends the Mercury interface definition language so that it separates a type's definition from its representation and permits multiple representations for a type. It also extends Mercury's registry to include names for representations. Less obvious, but just as important, is a mechanism for describing a context, i.e., the choice of representation for each type in a given interface. Context descriptions are complex because a user-defined type may be composed of several layers of abstraction, where types at each layer may themselves have multiple representations.

The bulk of this thesis is concerned with negotiating the representation that has the lowest cost for translating between internal and transmission representations. As will be seen later, it is very difficult to measure and compare quantitatively the relative costs from two applications. In contrast, programmers are often able to state qualitative cost "preferences" among contexts for a given application. Because user-defined types are composed of abstraction layers and distinct types may share components, a set of preferences typically forms a partial order, often with a complex structure. This introduces two problems: how do programmers specify sets of preferences, and is negotiation with preferences feasible?

The problem of specification is tackled by defining a language for stating preferences and a methodology for using the language. The methodology uses decomposition to divide an interface into smaller parts for which preferences can be stated more easily. The resulting preferences are then combined to form a set of preferences for the entire interface.

To show that negotiation is feasible I give a set of algorithms for constructing

data structures that correspond to preference expressions and for finding the context with the lowest translation cost given two sets of preferences.

This thesis makes the following contributions:

1. A framework for naming and describing both types and representations that makes a clear separation between the definition of a type and its representation.

2. A mechanism for describing contexts that contain data types composed of layers of abstraction.

3. A novel language for describing preferences together with a methodology for using the language.

4. Efficient algorithms for negotiating the context with lowest cost given two sets of preferences.

5. A practical design for integrating negotiation with preferences into real systems.

The rest of this thesis is divided conceptually into two parts. The first part introduces concepts and definitions and consists of Chapters 2 through 5. The second part is the "pragmatic" part that deals with implementation issues. It consists of Chapters 6 through 8. The detailed structure of the remainder of this thesis is as follows.

Chapter 2 introduces the concept of a vspace, which is the Mercury equivalent of a data type. It extends the Mercury interface definition language to permit the definition of user-defined vspaces with multiple representations. It concludes by defining extensions to the Mercury registry for naming.

Chapter 3 formally defines the key concepts of context, preferences, and what it means to negotiate the lowest-cost context.

Chapter 4 defines a language for describing preferences and a methodology for using the language.

Chapter 5 illustrates how one uses the language by giving detailed derivations of preference expressions for two example abstractions: a video display image and a VLSI mask.

Chapter 6 is the first chapter on pragmatics and describes the key algorithms, namely: how to "compile" preference expressions into the data structures used for negotiation, and how to compute the lowest-cost context given two sets of preferences. It analyzes several data structures for negotiation that were implemented and measured.

Chapter 7 shows how negotiation could be integrated into Mercury, OSI, and OSF/DCE.

Chapter 8 is the last chapter on pragmatics. It gives a design for the "presentation layer" of Mercury, showing that the interaction between negotiation and marshalling is relatively straightforward and incurs minimal overhead.

Chapter 9 is the conclusion. It summarizes the work, discusses alternatives to some of the design decisions, discusses related work, and points out future research directions.

Chapter 2

Vspaces, Representations and Negotiable Sets

2.1 Introduction

Just as individual program modules within a program can communicate by exchanging values, so too can individual programs in a heterogeneous system. However, there is an important difference in the way each passes values. A value within a program (usually) has a single representation that does not change when it is passed between two modules. Because the receiving module expects to perform certain operations directly on the value, the two modules share the implementation of the operations on that value. In many programming languages, a set of values that share a representation and a set of operations constitutes a type. In these languages, two program modules that share a type must also share an implementation for that type. Compilers perform type checking to ensure that program modules share types and their implementations correctly.

In contrast to program modules, two programs in a heterogeneous system need not share an implementation of the operations on a value in order to communicate that value. The only requirement with respect to correctness is that programs agree on the meaning of a value. In addition, we require that a specification of an interface between programs be independent of how values are represented in messages. This requirement allows us to separate the issues of correctness and implementation. It

also allows programs to negotiate the message representation of a value at runtime.

2.1.1 Vspaces

To satisfy the above requirements we group related values into sets called *vspaces*, to which we attach a specification and one or more representations. The specification gives the meaning of every value in the vspace. The term vspace underscores the difference between types in programming languages and "sets of values" in heterogeneous systems.[1] Like compilers and linkers for programming languages, heterogeneous systems provide binding mechanisms that ensure the correct sharing of vspaces among programs. Thus, the role played by vspaces in a heterogeneous system is similar to that played by types in a programming language.

In short, a vspace provides a common semantics for programs that use different internal representations for the same set of values and implement different operations on those values. Although vspaces are similar to types, they do not exist in any programming language, nor is it possible to write programs in terms of them. In order to communicate, programs must translate between types and vspaces. This requires that a vspace have a well-defined meaning and that its values can be represented in messages.

2.1.2 Abstract and Transfer Representations

In order to transmit a value it must have a bit-level representation within a message. Certain vspaces are considered primitive in that their representation is defined directly in terms of bit patterns, e.g., integers and booleans. However, this level of representation is too low-level to be specified easily by application-level programmers. The approach taken here is to divide the specification of the representation of a user-defined vspace into two levels: an abstract representation and a transfer representation. An *abstract representation* gives a structural specification of a vspace's representation in terms of other vspaces. It is a high-level definition of a representation and cannot be used directly in messages. A *transfer representation* gives a

[1]The use of vspace also has a practical aspect in that it prevents us from confusing "internal" program types with "external" vspaces, something that frequently occurs when discussing marshalling.

vspace's low-level representation for use in messages. A transfer representation for a primitive vspace is given directly by its definition, whereas a transfer representation for a user-defined vspace is derived methodically from its abstract representation.

One advantage of this two-level approach is that it frees programmers from the details of low-level representations, allowing them to concentrate instead on the high-level structure of a vspace. Another well known advantage of this approach is that we can write tools such as stub generators that generate code for translating between types and vspaces when given an abstract representation (e.g., see [Gib87, HL82, JRT85]).

2.1.3 Overview

As mentioned above, heterogeneous systems must provide mechanisms for ensuring that programs agree on the meaning of a vspace. In addition, programs must be able to negotiate the representation of a vspace at runtime. The model used in this thesis is that each vspace has a single definition that specifies its semantics as well as all possible representations. The representations for a primitive vspace are transfer representations, while those for user-defined vspaces are abstract representations. All vspace definitions are contained in a data repository called the *registry*. The specification of a vspace's semantics cannot change once the vspace is entered in the registry. As a result, programs always agree on a vspace's semantics. In contrast, new representations may be added at any time. Once added, they may not be deleted.

A programming environment based on this model is illustrated in Figure 2-1. As shown in the figure, the registry contains all definitions for vspaces and negotiable sets. Negotiable sets, which are defined later, are used to group vspaces into sets for the purpose of negotiation. In addition to containing definitions, the registry also assigns unique identifiers that a program uses to reference the defined entities when communicating with other programs.

As indicated by the arrows in the figure, the presentation compiler uses the definitions of vspaces and negotiable sets in the registry as input. As output, it produces data structures and code that a program uses to negotiate representations

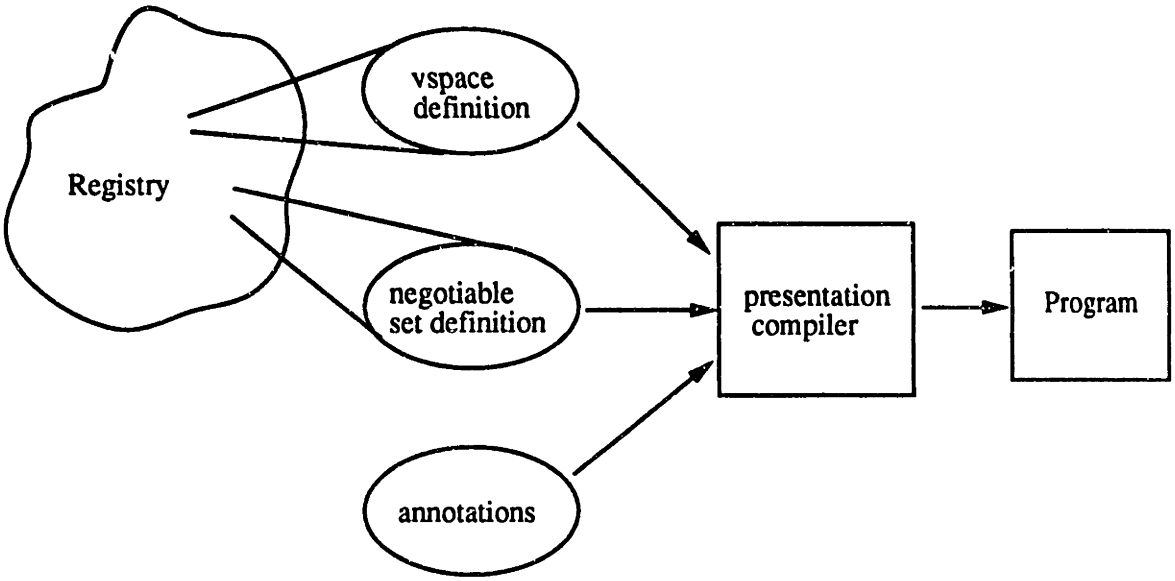


Figure 2-1: Overview of system environment

and communicate values. The compiler also uses program-specific annotations as input. Because annotations are specific to a given program, they are not contained in the registry. Chapter 7 discusses how annotations are used for negotiation. Annotations are also used to direct low-level details, such as how to represent a given vspace in the programming language used to implement the program. Such uses are outside the scope of this thesis.

The remainder of this chapter is organized as follows. Section 2.2 gives a very brief overview of what is meant by a specification. Section 2.3 introduces a language for defining vspaces and abstract representations. Section 2.4 introduces vspace components, which are used to specify the transfer representation for a given abstract representation. Section 2.5 describes negotiable sets. The requirements for a registry are given in Section 2.6. The chapter concludes with a brief summary.

2.2 Specifications

Every vspace has a specification that gives its meaning independently of any representation used for transmission. A specification may be either formal or informal. Examples of informal definitions are:

- int32 – the set of integers in the interval $[-2^{31}, 2^{31} - 1]$.
- int32_poly – the set of polynomials with int32-valued coefficients.

Formal definitions can give a precise meaning to values, for example, by the use of traits in Larch [GHW85a, LG86]. Formal specifications also have the advantage that a specific value in a given representation can be mapped into its formal meaning through the use of a (formal) abstraction function. As will be seen in the next section, even informal abstraction functions are important when specifying a representation for a user-defined vspace.

The identifiers int32 and int32_poly above are examples of names that a programmer can use to refer to the specified vspace. Names are important for checking that vspaces are used correctly and for referring to vspaces during negotiation. Before discussing naming, however, we need to look more closely at vspace and representation definitions because they provide mechanisms for introducing new names.

2.3 The Language Amalgam

Amalgam is a language for defining vspaces and abstract representations.[2] It includes a subset of the "Silver" language for describing vspaces in Mercury [Wei88]. In addition, Amalgam extends Silver with constructs for defining *abstract vspaces*, which are sets of values that can have more than one abstract representation. Like Silver, the purpose of Amalgam is to allow application programmers to define (possibly complex) vspaces using built-in and user-defined abstractions that are similar to those found in programming languages for defining types. As described here, Amalgam is intended to illustrate concepts and does not have all the features that one might have in a language intended for commercial use.

The description is divided into three sections, followed by a reference grammar. The first section describes the built-in vspaces and constructors. The second describes how programmers define abstract vspaces and constructors, and includes

[2] An amalgam is also an alloy of mercury with another metal, usually silver, that is solid or liquid at room temperature according to the proportion of mercury present.

an extensive example. The third describes equates, which allow programmers to introduce abbreviations for commonly used vspaces.

2.3.1 Built-in Vspaces and Constructors

Amalgam provides a number of built-in vspaces and vspace constructors that are used to build more complex vspaces.[3] Among them are the following scalars:

null - the set containing a single value, which is denoted by **nil**.

bool - the set containing two values, which are denoted by **true** and **false**.

char - the set of ASCII characters.

byte - the set of non-negative integers in the range $[0, 2^8 - 1]$.

card16 - the set of non-negative integers in the range $[0, 2^{16} - 1]$.

card32 - the set of non-negative integers in the range $[0, 2^{32} - 1]$.

int16 - the set of integers in the range $[-2^{15}, 2^{15} - 1]$.

int32 - the set of integers in the range $[-2^{31}, 2^{31} - 1]$.

float32 - the set of single-precision floating-point values defined by IEEE standard 754-1985.

There are also the following vspace constructors:

fixed_sequence[n:**int**, V:**vspace**]

A value s of this vspace is a sequence of exactly n values of vspace V. The i^{th} element of s is denoted by $s[i]$ where $1 \leq i \leq n$.

bounded_sequence[n:**int**, V:**vspace**]

A value s of this vspace is a sequence of at most n values of vspace V. The actual length of s is denoted by $length(s)$, and the i^{th} element of s is denoted by $s[i]$, where $1 \leq i \leq length(s)$.

struct[id_1:V_1, ..., id_k:V_k]

A value of this vspace is a sequence of k values, the first of vspace V_1, the second of vspace V_2, and so on. Case is irrelevant in the field names (i.e., the id_i). In addition, the order of the fields is important, so struct[a:V1,

[3]The descriptions of scalars and constructors follow closely those found in [Wei88].

b:V2] and struct[b:V2, a:V1] are different vspaces. If s is a struct value containing the field *id*, then the value of field *id* is denoted by *s.id*.

oneof[id_1:V_1, ..., id_k:V_k]

A value of this vspace is a pair consisting of a tag (one of the identifiers id_i) and a value, where the value is a member of the vspace corresponding to the tag. Like structs, the case is irrelevant in tags. In contrast to structs, the order of the tags is not important. A oneof vspace has associated with it a set of functions of the form is_id_i and $value_id_i$. If s is a oneof value having a tag *id* with associated vspace V, then $is_id(s)$ is true if and only if the tag of s is *id*. It is false otherwise. If $is_id(s)$ is true, then $value_id(s)$ denotes a value of V. If $is_id(s)$ is false, then $value_id(s)$ is undefined.

container[V:**vspace**]

Containers are similar to references in that they allow values to share structure within a message (e.g., a directed graph). Their semantics are discussed in [Wei87a].

The functions $length(s)$, $s[i]$, *s.id*, $is_id(s)$ and $value_id(s)$ in the above specifications are provided so that they may be used in specifications for vspaces built from the corresponding constructors. They are not a part of the language per se because Amalgam does not specify a syntax for specifications.

Note also that the specification for each of the above constructors is given in terms of the constructor's parameters. When a constructor is instantiated by specifying actual values for each parameter, the result is a specification for the constructed vspace. In other words, the specification is actually a "specification constructor."

The built-in scalars and constructors described above are referred to as *base vspaces* and *base constructors* because they form the basis for building user-defined vspaces.

2.3.2 Abstract Vspaces and Constructors

User-defined vspaces in Amalgam are called *abstract* vspaces. Abstract vspaces are one of the most important features of the language because they allow programmers to define new vspaces with multiple abstract representations. A value of an abstract

vspace is referred to as an *abstract value.* An abstract vspace definition has the following form:

$id_v[id_1:kind_1, \ldots, id_n:kind_n]$ = **abstract**
  %specification
  rep id_{r1} = *vspace-expression*
    %representation-invariant
    %abstraction-function
  ⋮
  rep id_{rk} = *vspace-expression*
    %representation-invariant
    %abstraction-function
  end

The keyword **abstract** in the above form identifies it as an abstract definition with id_v as the name of the new abstract vspace. The parameter list $[id_1:kind_1, \ldots, id_n:kind_n]$ is optional and, if present, means that the definition is for an abstract constructor. The identifiers in the parameter list may occur in the vspace expression of a representation defined in the body. Each parameter has a "kind" that specifies the class of values to which the identifier may be bound. Currently, there are only two kinds, which are denoted by the keywords **vspace** and **int**.

The specification comes first and takes the form of a comment because Amalgam does not require the use of a particular specification language and, therefore, does not interpret the specification. Comments are introduced by a percent sign.

The specification is followed by one or more representations, each of which is introduced by the keyword **rep**. A representation is simply another vspace and, therefore, gives a structural definition of the abstract vspace in terms of other vspaces. Definitions can be recursive in that the name of the abstract vspace can appear in a representation. Mutual recursion among abstract vspaces is also allowed. In the case of an abstract constructor, the identifiers id_i in the parameter list may also occur in the representation.

A representation for an abstract vspace does not specify a transfer representation because it consists of other (possibly abstract) vspaces. Therefore, a representation

is still abstract as far as communicating values is concerned; it is just at a lower level of abstraction than the abstract vspace it represents.

Each representation has a name, a representation invariant and an abstraction function. The name distinguishes one representation from another and provides a handle that can be used for negotiation. The *representation invariant* specifies the subset of values from the representation vspace that actually represent values in the abstract vspace. The *abstraction function* describes the correspondence between representation values and abstract values. Typically, it takes the form of a mapping from representation values to abstract values. More information on the use of representation invariants and abstraction functions can be found in [LG86]. Like specifications, representation invariants and abstraction functions are given as comments.

To use an abstract constructor (e.g., in the representation of some other abstract vspace), it must first be instantiated by providing a value for each of its parameters. If id_c is the name of an abstract constructor with formal parameters id_1:k_1 through id_n:k_n, then an *instantiation* of id_c is denoted by $id_c[a_1, \ldots, a_n]$, where each actual parameter a_i has kind k_i. Actual parameters of kind **vspace** are denoted by vspace expressions, for which a grammar is given later. Parameters of kind **int** are denoted as usual by numerals.

The form and meaning of an abstract vspace definition are best illustrated by an example. Figure 2-2 shows a definition of an abstract constructor for a two dimensional matrix. The specification consists of the three functions *num_rows*, *num_cols*, and $a[i, j]$, which interpret a given abstract value. There are two representations corresponding to row major and column major form. In both cases the representation invariant specifies the same set of legal values, namely, those that have a non-zero value for the num_rows and num_cols fields, and have a data field with a length equal to the number of columns times the number of rows. Values that do not meet these restrictions are nonsense values because they do not represent any matrix as defined by the specification. Transmitting a representation value that does not meet the representation invariant is considered an error.

The abstraction function specifies the mapping between elements of the data field and elements of the abstraction. The only difference between the two repre-

```
matrix_2d[V:vspace] = abstract
    %
    % A value of this space is a two-dimensional m by n matrix whose elements
    % are values of vspace V, where 0 < m < 2^32, 0 < n < 2^32 and n × m < 2^32.
    % A value a of this vspace is defined by the following three functions:
    %
    %       num_rows(a) = the number of rows in a.
    %       num_cols(a) = the number of columns in a.
    %       a[i, j] = the element in the j^th column of the i^th row of a,
    %         where 1 ≤ i ≤ num_rows(a) and 1 ≤ j ≤ num_cols(a).
    %
    rep colmaj = struct[num_rows:card32, num_cols:card32,
                        data:bounded_sequence[2 ** 32, V]]
        %
        % rep-invariant: 0 < r.num_rows and 0 < r.num_cols
        %                and length(r.data) = r.num_rows × r.num_cols
        % abstraction function:
        %       num_rows(r) = r.num_rows
        %       num_cols(r) = r.num_cols
        %       r[i, j] = r.data[(i − 1) × r.num_cols + j]

    rep rowmaj = struct[num_rows:card32, num_cols:card32,
                        data:bounded_sequence[2 ** 32, V]]
        %
        % rep-invariant: 0 < r.num_rows and 0 < r.num_cols
        %                and length(r.data) = r.num_rows × r.num_cols
        % abstraction function:
        %       num_rows(r) = r.num_rows
        %       num_cols(r) = r.num_cols
        %       r[i, j] = r.data[(j − 1) × r.num_rows + i]

    end
```

Figure 2-2: Example of an abstract vspace definition.

sentations is that the role of rows and columns is exchanged. This example also demonstrates the importance of abstraction functions in the definition of a representation. Without the abstraction function there would be no visible difference between the row major and column major representations. Therefore, even though it may be described informally, the abstraction function is the primary method for assigning a distinct interpretation to a representation.

The following examples illustrate possible instantiations of matrix_2d:

matrix_2d[**int32**]

matrix_2d[**struct**[x:**float32**, y:**float32**]]

matrix_2d[matrix_2d[point]], where "point" is an identifier denoting a vspace.

2.3.3 Equates

Amalgam also provides an equate construct for defining abbreviations. An equate has the form:

$id_e[id_1:kind_1, \ldots, id_n:kind_n]$ = *vspace-expression*

An equate associates the meaning of the vspace expression on the right-hand-side with the identifier id_e. It may also have an optional parameter list, in which case it acts like a constructor in that the id_i appear in the vspace expression. The following example defines an equate that is used frequently in later chapters:

sequence32[v:**vspace**] = **bounded_sequence**[2 ** 32,v]

An equate constructor is instantiated in the same way as an abstract constructor, e.g., sequence32[**int32**]. Equate definitions, like abstract vspace definitions, may also be recursive or mutually recursive.

Unlike an abstract vspace, an equate does not introduce a new vspace name whose meaning is distinct from the vspace that defines it. For example, the expression sequence32[**int32**] denotes the same vspace as **bounded_sequence**[2 ** 32,**int32**]. This difference between equates and abstract vspaces is important when "type checking" the use of vspaces; equates are checked using structural equivalence while abstract vspaces use name equivalence. Further discussion about type checking can be found in [LW86, Wei87a, Wei87b, Wei88].

Vspace-exp → *id* | *constructed*
constructed → *instantiation*
| **struct**[*field*$_1$: *Vspace-exp*$_1$, ..., *field*$_n$: *Vspace-exp*$_n$]
| **oneof**[*tag*$_1$: *Vspace-exp*$_1$, ..., *tag*$_n$: *Vspace-exp*$_n$]
instantiation → *id*[*argument*$_1$, ..., *argument*$_n$]
argument → *Vspace-exp* | *integer-numeral*

Vspace-def → *equate* | *abstract*
equate → *id* [param-list] = *Vspace-exp*
abstract → *id* [param-list] = **abstract** *specification* { *abstr-rep* }$^+$ **end**
abstr-rep → **rep** *id* = *Vspace-exp invariant abstr-function*
param-list → [*id*$_1$: *kind*$_1$, ..., *id*$_n$: *kind*$_n$]
kind → **vspace** | **int**
comment → % *text end-of-line*

Figure 2-3: Grammar for vspace expressions and definitions

2.3.4 Reference Grammar

A reference grammar summarizing Amalgam's concrete syntax is given in Figure 2-3. This syntax will be used in examples throughout the rest of this thesis. Non-terminals in the grammar are shown in italics and keywords in boldface type. The meta-character '|' is used to separate alternative productions for the same non-terminal. The bold-faced set notation " { }$^+$" indicates one or more occurrences of the material between the braces. Bold-faced square brackets " [ ]" indicate an optional occurrence of the material between the brackets. Non-bold-faced characters such as ':', '[', ']', and ',' are terminals. The three dots "..." between constructs indicate a comma-separated list of one or more instances of those constructs. Finally, a comment is introduced by a percent sign and consists of all the text between the percent sign and the end of the line.

Note that the scalar vspaces and the constructors for fixed sequences, bounded sequences and containers are not listed in the grammar because they are covered under the case of identifiers and instantiations. They are considered to be abstract vspace definitions that are provided by a "standard" environment. The constructors for struct and oneof cannot be handled in this way because Amalgam is not powerful enough for users to define constructors that take variable-length field lists

as arguments.

2.4 Vspace Components

A value must be given a transfer representation in order for it to be sent from one program to another. Recall from Section 2.1.2 that transfer representations are derived from abstract representations. The exact details of how they are derived must wait until Chapter 3. This section gives a brief sketch of how one derives a transfer representation to motivate the definition of vspace components, which are Amalgam-specific constructs.

Given an abstract representation expressed in Amalgam that consists of only base vspaces and constructors, it is relatively straightforward to generate a transfer representation by choosing a transfer representation for each base vspace and constructor appearing in the expression. In the case of a base vspace, the transfer representations are provided as primitives. For example, for **int32** we may choose a 32 bit, two's complement representation with the most significant byte transmitted first. In the case of a base constructor, the transfer representation is really a "representation constructor" because it describes how to generate a transfer representation when given a transfer representation for each parameter. For example, for a **struct** we simply concatenate the transfer representations of the fields in the order they are specified in the abstract representation.

Intuitively, the method just described specifies a transfer representation by structural induction. The base vspaces correspond to the induction basis because their transfer representations are defined as primitives. The constructors correspond to induction steps. By specifying a transfer representation for all the base vspaces and constructors of Amalgam, we can generate a transfer representation for any vspace, as long as it does not contain any abstract vspaces or constructors.

In the presence of abstract vspaces and constructors the approach is similar, i.e., we choose a representation for each abstract vspace and constructor that appears in an expression. However, in this case the representations are abstract, and we must derive transfer representations for them. This is covered in Chapter 3. What is needed here is to define all of the "entities" in an expression for which we must

choose a representation. These entities are called *vspace components*. In Amalgam, a vspace component is either a base vspace, base constructor, abstract vspace, or abstract constructor. These are the only constructs in Amalgam that introduce multiple representations.

In general, I will use a fixed-width font to denote vspace components, e.g., `oneof` denotes a vspace component, whereas **oneof** is a reserved identifier that denotes a base vspace constructor.

Representations for components must be named so that they can be referenced by transfer-representation specifications. The names for abstract vspaces and constructors are given by the **rep** clauses in their definition. The names for base vspaces and constructors will be introduced as they are needed in examples. Issues of name resolution are discussed in Section 2.6.

2.5 Negotiable Sets

For the purpose of negotiation, vspaces are grouped into sets called *negotiable sets*. Informally, negotiation is the process by which two presentation peers agree on a representation for the vspaces in a given negotiable set. The details of negotiation and how one specifies the representation of a set of vspaces are left to later chapters. At this point the concept of a negotiable set is a rather simple entity to which we attach no further semantics. It could consist of a single vspace, the set of vspaces in a single message or those in a set of messages. By making it a general concept we gain the flexibility needed to apply it to different systems. Chapter 7 shows how negotiable sets can be used in real systems such as Mercury and OSI.

The only requirements for negotiable sets are that the presentation layer be able to name them unambiguously in presentation protocol messages and that two presentation entities agree on their definition. Therefore, I assume that the registry provides mechanisms for defining and naming negotiable sets. In order to maintain flexibility, these mechanisms should be orthogonal to any other mechanism for grouping vspaces, e.g., for the purpose of controlling the scope of names or encapsulating a set of related vspace definitions.

2.6 The Registry

The previous sections have defined a number of named entities that are shared by programs and programmers, namely vspaces, vspace components, representations, and negotiable sets. It is essential that all these entities have the same meaning everywhere. To meet this requirement I assume the existence of a *registry* that coordinates the creation and sharing of definitions and the assignment of names to entities. The exact form of the registry is not important as long as it meets the following requirements:

1. Provides a repository for definitions.
2. Provides a programmer-friendly name space.
3. Allows new representations to be added to the definition of an existing vspace component.
4. Assigns unique identifiers to negotiable sets.
5. Assigns unique identifiers to vspace components.
6. Assigns identifiers to representations for a given component.
7. Is scalable in size.

These requirements are discussed below, together with a brief discussion of how the registry is used.

The registry must provide a repository for definitions simply because definitions are shared; a repository not only makes them available, but also ensures that all users see a consistent definition. It is also desirable to have a single reference language for specifying definitions (e.g., a suitable extension of Amalgam) to prevent users from specifying conflicting definitions in different languages for the same entity. Other definition languages could be allowed as long as there is an unambiguous translation between them and the reference language.

The need for a programmer-friendly name space comes from our expectation that the registry will be large. Programmers will need a means for navigating through the registry's name space: In fact, I assume there is some naming mechanism that allows

definitions to be grouped into named sets similar to the way files in a file system are grouped into named directories. The names within a set form an *environment* for resolving identifiers in Amalgam definitions that are contained within that set. References between sets allow names defined in one environment to be visible in another.

The ability to define new representations for existing vspace components is required so that systems can evolve. It is also important that existing definitions not be modified (including deletion) because it is impossible in general to tell if there is a program that depends on the original definition.

So far, all the names for defined entities such as vspace components and representations have been given as character strings (e.g., identifiers in Amalgam). These names cannot be used by a presentation layer to negotiate representations because their meaning is relative to an environment. Therefore, the registry must assign each entity an identifier (ID) that is independent of any name used by programmers.

A unique identifier (UID) must be assigned to a negotiable set because negotiable sets are the primary objects of reference for negotiation.

UIDs are required for vspace components because they can appear as components in more than one negotiable set. Making the identifier of a vspace component relative to a negotiable set would introduce aliases, where different UIDs would denote the same component. This could make it difficult to extend negotiation in the future.

Identifiers for component representations need be distinct only with respect to the vspace component that they represent. This is because a given representation is valid for exactly one vspace component. If necessary, a UID for a representation can be constructed by concatenating the representation's relative ID to its vspace component's UID. The registry can take advantage of this property to assign small identifiers to representations.

In the rest of this thesis I will assume that UIDs for vspace components and relative IDs for representations have a simple structure such as 64-bit unsigned integers. Requiring systems to use such simple identifiers is probably too strong in general, although there are systems that do so [DLM+87]. However, because identifiers for vspace components and representations are communicated and manipulated during negotiation, this assumption allows efficient implementations for marshalling, com-

paring, and indexing. Changing this assumption would affect the performance of the negotiation algorithms described in Chapter 6.

Finally, the registry must be scalable over a wide range of sizes from a few hundred vspace definitions to potentially millions of definitions. The issues involved in designing such a registry are beyond the scope of this thesis.

2.7 Summary

A vspace is a set of values that has a name, a specification and one or more representations. A representation is divided into two levels: an abstract representation and a transfer representation. Abstract representations are described using the Amalgam language. Amalgam provides a number of base vspaces and constructors that allow programmers to define more complex vspaces. A vspace that is constructed directly from base vspaces and constructors has only one abstract representation. Amalgam also allows programmers to define an abstract vspace, which is a new vspace in that it is considered distinct from all other vspaces. Abstract vspaces also allow programmers to introduce vspaces that have multiple abstract representations.

An abstract representation for a vspace is constructed out of vspace components. A vspace component in Amalgam is either a base vspace, base constructor, abstract vspace, or abstract constructor. Vspace components provide the means for introducing multiple representations. A vspace's transfer representation is defined inductively by specifying a representation for each vspace component in the vspace's abstract representation. In the case of an abstract component, the representation is an abstract representation for which a transfer representation must be specified in turn.

The presentation layer of an application negotiates representations for sets of vspaces called negotiable sets. Negotiable sets are a general concept that can be applied to different systems.

Finally, the registry provides the means by which programs and programmers share names and definitions. It not only acts as a repository for definitions, but also assigns UIDs. It also allows the creation of environments that control the scope of names.

Chapter 3

Contexts, Preferences and Negotiation

In order to communicate values of the vspaces in a negotiable set, two application-level peers must agree on the transfer representation for each vspace in that set. Informally, a specification of the transfer representation for all the vspaces in a given negotiable set is called a "context." Because any vspace can have multiple transfer representations, there will likely be many possible contexts for a negotiable set. An application entity *supports a context for a negotiable set* if it can both encode and decode every value of every vspace in the negotiable set using the transfer representation specified by that context. Therefore, in order to communicate, two peers must agree on a context that they both support. In addition, they must coordinate the use of that context so that each peer knows the transfer representation of every value in a message. The process of agreeing on a context is called *negotiation* and involves exchanging information at runtime describing the contexts that each peer supports. The primary interest of this thesis is not how to negotiate just any context, but to negotiate the "best" context with respect to some cost measure.

The remainder of this chapter is devoted to defining precisely what is the best context for a negotiable set with respect to two peers. It focuses on how one compares contexts, what information is exchanged during negotiation, and what it means for two peers to agree on the best context. It is concerned with neither the negotiation protocol that governs the exchange of information nor the mechanism that

coordinates the use of contexts with the transmission and reception of data.

The presentation is divided into three parts. The first gives a formal definition of a context. The second focuses on how one compares two contexts with respect to cost. As will be seen, a quantitative comparison of two contexts is often difficult. Despite this fact, it is possible for programmers to state qualitative preferences between contexts that respect differences in cost. The approach presented is that the programmer of a peer application specifies a set of preferences among the contexts that the peer supports for a given negotiable set. This set of preferences is called a context list and is defined formally in Section 3.2 below. The last part, Section 3.3, describes how to take a context list from each of two peers and compute the best context.

3.1 Contexts

Informally, a "context" is a specification of the transfer representation of every vspace in a given negotiable set. As described in Section 2.4, given an Amalgam expression that defines an abstract representation for a vspace, one specifies a transfer representation for that vspace by choosing a representation for each vspace component that appears in the expression. In the case where a component is an abstract vspace or constructor, the component's representation is another abstract representation. In such cases it is necessary to specify a transfer representation for this abstract representation. Of course, the situation will repeat itself in a top-down manner when there are several layers of abstraction. At each layer the choice of representation determines the components of the next lower layer. The specification is complete when the transfer representation of every layer has been specified.

As an example consider the vspace matrix_2d[**int32**] from the previous chapter. If we choose the `rowmaj` representation for the abstract component `matrix_2d`, then we have to specify a representation for the components `bounded_sequence` and `card32` that occur in the abstract representation defined for `rowmaj`. Because there are no abstract components in this layer, we only need to specify a representation for the component `int32` to complete the specification for the vspace.

The formal definition of a context requires a fair amount of machinery to capture

the informal intuition above. Therefore, the definition is given in three steps below. The first step introduces the function *components*, which computes the vspace components in an Amalgam expression. The second introduces the concept of an assignment, which formalizes specifying a representation for each vspace component in a list. The third and last step formally defines a context.

3.1.1 The Function *Components*

A decision that must be made at this point is whether or not to allow different transfer representations for distinct occurrences of the same vspace component in a vspace expression. For example, given the expression **struct**[a:**int32**, b:**int32**], could we have different representations for the two integers? In this case the answer is probably not, because integers usually have only one representation on a given CPU. On the other hand, if the fields "a" and "b" were rectangles, then the answer might be yes. The problem with allowing different representations is that it introduces complexity. Not only is the structure of a context more complex, but so is negotiation. For example, what if one program uses the same representation for both but the other uses a different representation for each one? Must we negotiate each occurrence separately?

The simple solution, which is the solution used here, is to require that all occurrences of the same vspace component use the same representation. This solution is the appropriate choice in most cases because programs typically use one internal representation for all values of a given type. Using the example above where the fields are rectangles, it is most likely the case that the program uses the same type to represent both instances of the rectangle. In this case, the relative costs of translation will be the same for both fields.

In order to define a transfer representation for an abstract representation expressed in Amalgam, we must know the set of vspace components that occur in the expression. The function *components* computes this set and has the signature *Vspace-exp* $\rightarrow$ *Set*[*vspace-component*]. The function itself is defined by the following

structurally recursive formula:

$$components(id) = \begin{cases} \emptyset, & \text{if } id \text{ is a variable} \\ \{id\}, & \text{if } id \text{ denotes a vspace component} \\ components([\mathbf{null}/id]def(id)), & \text{if } id \text{ denotes an equate} \end{cases}$$

$$components(c[a_1, \ldots, a_n]) = \{c\} \cup \left(\bigcup_{i \in vargs} components(a_i) \right) \quad (3.1)$$

$$\text{where } i \in vargs \Leftrightarrow kind(a_i) = \mathbf{vspace}$$

$$components(\mathbf{struct}[id_1 : e_1, \ldots, id_n : e_n]) = \bigcup_{1 \le i \le n} components(e_i)$$

$$components(\mathbf{oneof}[id_1 : e_1, \ldots, id_n : e_n]) =$$

$$\{\mathtt{oneof}\} \cup \left(\bigcup_{1 \le i \le n} components(e_i) \right)$$

For the most part, the components of an expression are given by taking the union of the components of each subexpression. When the expression is an identifier that denotes a vspace component, then the result consists of just that component. There are a few special cases that require some explanation.

The first such case is the subcase of an identifier where the identifier is a formal parameter variable that appears within the definition of a constructor. In this case, the components of the corresponding actual parameter are computed by the second clause of equation 3.1, which computes the components of an instantiation. Therefore, there are no components for the formal paramenter.

The subcase where an identifier denotes an equate is a little complex because it may be recursive. The notation $[x/id]\mathcal{E}$ means "substitute the expression x for the identifier id in the expression $\mathcal{E}$." The function $def(id)$ gives the expression associated with equate id. The substitution of **null** for id above is used to terminate any recursion. With this substitution, the components of an equate are computed only once. Of course, it is possible to define *components* without making this substitution, but it would involve defining a fixed-point semantics, which is much

more complex.

The last case that requires an explanation is that for **struct**. In this case there is no struct vspace component in the result. The reason for this is that a struct has only one transfer representation, namely the concatenation of the transfer representation of each field in the order specified. This reflects the sematics of structs, which states that order is important.

Once we have the vspace components in an expression, we must specify a representation for each to obtain a transfer representation. This is the purpose of an assignment, which is the next step in the definition of a context.

3.1.2 Assignments

The introduction of assignments at this point is actually more than just a convenience for defining a context. Assignments can also be thought of as subsets of a context. As will be seen in Chapter 4, assignments are the basic building blocks for expressing preferences among contexts.

An *assignment* is a partial function from vspace component UIDs to representation UIDs that maps a given vspace component to a representation for that component. Every assignment a has a *signature*, denoted by $sig(a)$, that gives the set of vspace components for which a is defined. If u is a vspace component UID and $u \in sig(a)$ then $a(u)$ denotes a representation UID for u.[1] If $u \notin sig(a)$ then $a(u)$ is undefined.

An assignment is denoted by an expression of the form

$$\{comp\text{-}id_1 : rep\text{-}id_1, \ldots, comp\text{-}id_n : rep\text{-}id_n\}$$

where $comp\text{-}id_i$ is an identifier denoting a vspace component UID and $rep\text{-}id_i$ is an identifier denoting a UID of a representation for that vspace component. If a is an assignment denoted by the above expression then $a([\![comp\text{-}id_i]\!]) = [\![rep\text{-}id_i]\!]$, where $[\![id]\!]$ gives the UID denotated by the identifier id. As described in Section 2.6, the denotation of an identifier is determined by a compile-time environment that is provided by the registry. The presentation compiler is responsible for looking up

[1] Because representations are defined in the registry and new representations may be added, the range of $a(u)$ is actually a function of time. This dependence can be ignored in practice.

the denotation of each identifier in the registry and checking that the mapping for each component is valid.

There are three definitions that are needed before giving a formal definition of a context. The first is the predicate $abstract(u)$, which is true if and only if the vspace component with UID u is an abstract vspace or abstract constructor.

The second definition is the predicate $in_rep_of(u_1, u_2, a)$, which specifies when a vspace component u_1 is in the representation of an abstract vspace component u_2 with respect to an assignment a. It is defined by the following recursive formula:

$$\begin{aligned} &in_rep_of(u_1, u_2, a) \Leftrightarrow \\ &\qquad u_2 \in sig(a) \wedge abstract(u_2) \\ &\qquad \wedge((u_1 \in components(a(u_2))) \\ &\qquad\quad \vee(\exists\, u_3 \in sig(a) \text{ s.t. } in_rep_of(u_1, u_3, a) \wedge in_rep_of(u_3, u_2, a))) \end{aligned} \tag{3.2}$$

which says that u_1 is in the representation of u_2 with respect to a if and only if u_2 is in the signature of a, u_2 is an abstract component, and u_1 occurs either directly or indirectly in u_2's (abstract) representation. It is related to the concept of abstraction layers. Intuitively, u_1 is in the representation of u_2 if u_1 is a component of some abstraction layer in the representation of u_2, where the (abstract) representations of any intermediate layers are given by a.

The third definition is the predicate *is_complete*, which determines if an assignment completely specifies the transfer representation for a vspace. In the case where a vspace expression e contains an abstract component, choosing a representation for every component in $components(e)$ is not sufficient to specify a transfer representation. For example, consider e = matrix_2d[**int32**], in which case $components(e)$ = {**matrix_2d**, **int32**}. Then the assignment {**matrix_2d:rowmaj**, **int32:be**}, where **be** denotes the "big-endian" representation, does not specify a transfer representation because the **rowmaj** representation is an abstract representation. In order to be complete, the assignment must specify representations for the components of the **rowmaj** representation. If any of those components are abstract, then the process is repeated until there are no abstract components left unspecified. This leads to the

following definition, where a is an assignment and e is a vspace expression:

$$\begin{aligned} &\mathit{is_complete}(a, e) \Leftrightarrow \\ &\qquad (\mathit{components}(e) \subseteq \mathit{sig}(a)) \\ &\qquad \wedge(\forall u \in \mathit{sig}(a) \,.\, \mathit{abstract}(u) \Rightarrow \mathit{components}(a(u)) \subseteq \mathit{sig}(a)) \end{aligned} \tag{3.3}$$

Like *in_rep_of* above, *is_complete* is related to the concept of abstraction layers. An assignment completely specifies a transfer representation for a vspace if and only if it specifies a representation for every layer of abstraction.

3.1.3 Definition of a Context

We are now ready for the definition of a context for a negotiable set. Let E be a set of vspace expressions, S the negotiable set consisting of the vspaces denoted by the elements of E, and U the set of vspace components $\{u \mid u \in \mathit{components}(e), e \in E\}$. Then the predicate $\mathit{is_context}(c, S)$, read as *c is a context for S*, is defined by the following:

$$\begin{aligned} &\mathit{is_context}(c, S) \Leftrightarrow \\ &\qquad \forall e \in E \,.\, \mathit{is_complete}(c, e) && \text{[complete]} \\ &\qquad \wedge(\forall u_1 \in (\mathit{sig}(c) - U) \,.\, \exists u_2 \in U \text{ s.t. } \mathit{in_rep_of}(u_1, u_2, c)) && \text{[necessary]} \end{aligned} \tag{3.4}$$

The first line of the definition states that a context must specify a transfer representation for every vspace in S. The second line says that every component that does not appear directly in some vspace in S is required for specifying the transfer representation of some vspace in S.

The requirement for completeness is quite obvious. However, the requirement for necessity is less so. The reason for it is that negotiation involves sets of contexts and we must ensure that distinct contexts differ only in components that are relevant to the vspaces in the negotiable set. This is because there is a cost associated with each context, and this cost should reflect only the costs incurred by the negotiable set. There is also a practical matter. Algorithms that compare contexts from two peers need not be concerned about irrelevant information in the contexts. This can lead to a substantial saving because, otherwise, an algorithm would have to compute the necessary components of a context before comparing it with another in order to eliminate extraneous components.

As a final remark, note that a context maps a component to exactly one representation. If a context specifies representations for the components of two vspaces and those vspaces share a component, then they also share the representation. For example, consider the assignment {`matrix_2d:rowmaj`, `int32:be`, `float32:ieee`} that specifies part of a context for the negotiable set consisting of the vspaces matrix_2d[**int32**] and matrix_2d[**float32**] from Chapter 2. In this case both matrices use the row major representation. It is not possible to have one use the row-major representation and the other column-major. This would also be the case if the negotiable set contained the single vspace consisting of a struct where one field was matrix_2d[**int32**] and another matrix_2d[**float32**].

The restriction that vspaces must use the same representation for common components was made in order to simplify the specification of contexts and the computation necessary for negotiation. There is a trade-off between the complexity of the structure of a context, the cost to negotiate a context, and the context's ability to describe the optimal representation for every vspace. This issue will be revisited in Chapter 9, after the costs for negotiation have been analyzed in Chapter 6

This completes the formal definition of contexts. We are now ready to tackle the problem of how to associate costs with contexts and compare them.

3.2 Preferences and Context Lists

This section addresses the problem of assigning preferences to the set of contexts supported by a single peer for a given negotiable set. What we would like is a quantitative measure that assigns a cost to each context. In this case, one context is preferred to another if it has a lower cost. An example cost measure is the average number of CPU cycles needed to encode and decode each message. Given this measure, it would be simple to order all the contexts for a negotiable set. However, as I will show below, it is difficult to produce a quantitative cost measure. In fact, it may not be possible to order two contexts given the information available. After discussing the problems with quantitative cost measures I will describe an approach based on stating qualitative preferences. At the end of this section I will define formally a context list. A context list captures a peer's preferences for the

contexts of a negotiable set and constitutes the information exchanged by peers during negotiation.

3.2.1 Problems with Quantitative Measures

The following example will help illustrate the problems with constructing a model based on a quantitative cost measure. Consider a server that performs arithmetic on two-dimensional matrices of thirty-two bit integers and thirty-two bit floating point numbers. Assume that the negotiable set contains two vspaces, matrix_2d[**int32**] and matrix_2d[**float32**], where matrix_2d[v] is the abstract vspace constructor from Chapter 2. The set of vspace components for this negotiable set consists of the components `matrix_2d`, `int32`, and `float32`. No matter which representation is chosen for `matrix_2d`, a complete assignment must also contain the components `bounded_sequence` and `card32`. Therefore, any context must specify representations for each of these five components.

For the sake of simplicity we assume that `bounded_sequence` has only one representation. Furthermore, we assume that the matrices are large enough that the cost of marshalling the values for the fields "num_rows" and "num_cols" is dominated by the cost of marshalling the matrix elements. This allows us to focus on the costs related to the choice of representation for the components `matrix_2d`, `int32`, and `float32`.

Now, assume that the vspace `int32` has two representations, `be` for big-endian and `le` for little-endian, and that `float32` has two representations, `ieee` for IEEE format and `vax`, which is the representation on a Vax[2] CPU.[3] If we focus on just the three major components, there are eight possible assignments corresponding to the eight combinations of representations for the three components.

In order to compute a cost measure we need to know the internal representation used by the application for the vspaces matrix_2d[**int32**] and matrix_2d[**float32**]. For this example we assume that the peer is running on a Vax, so the internal representation is little-endian for integers and vax for floating-point. In addition,

[2]Vax is a trademark of the Digital Equipment Corporation.

[3]In a real system we would have to worry about the byte order of the IEEE format. This is ignored for the sake of simplicity.

we assume that the internal representation of matrices corresponds to that given by the row-major abstract representation, which would be the case if the application were written in C.

We also need to know something about the code generated by the presentation compiler. Assume that the compiler generates in-line code for converting between the internal representation and both transfer representations for integers and floats. This is a reasonable assumption because there are compilers that do so today. In the case where the abstract representation for `matrix_2d` is row-major, the presentation compiler can generate the marshalling routines because the abstract representation matches the internal representation. In the case of column-major, assume that the programmer provides a separate procedure to transpose a matrix. This procedure is called by the code generated by the presentation compiler. This would be the case for a C program in the Apollo system [DLM+87]. Note that this case differs from that in Chapter 1 because the transposition procedure is not hand-written to use the low-level RPC system interface directly.

In summary, there are eight marshalling routines corresponding to the eight assignments. Four routines correspond to the case where the transfer representation is row-major. These routines are completely generated by the presentation compiler. The remaining four routines correspond to the case where the transfer representation is column-major. Although these routines are generated by the presentation compiler, they must call the programmer-supplied procedure to transpose the matrix.

This concludes the description of the example. We are now ready to look at the problems.

The first problem is that the presence of programmer-supplied code in the marshalling routines for abstract vspaces makes it difficult to obtain an accurate model for the cost of encoding and decoding data. In the above example the marshalling code for the column-major representation of `matrix_2d` is provided by the programmer. It most likely is written in a high-level programming language. In general, neither the programmer nor the compiler can specify an accurate formula that would give the cost in, say, CPU cycles per element. It might be possible to profile the code using representative data sets and extract a model from the results, as was

done for the code profiled in Chapter 1. However, it is unreasonable to expect the programmer to do this for every combination of representations for all vspaces.

The second problem is that there are several significant sources of uncertainty that must be reflected in the model. The first source is the lack of accuracy mentioned in the previous paragraph. To illustrate this consider the following two contexts for our example negotiable set:

{`matrix_2d:rowmaj`, `int32:be`, `float32:vax`} and
{`matrix_2d:colmaj`, `int32:le`, `float32:vax`}.

The first assignment matches the internal representation for `matrix_2d` but not for `int32`, whereas the second is exactly the opposite. In the first there is a penalty for accessing the elements in the wrong order, namely, a separate transposition step. In the second there is a penalty for swapping bytes. In each case the penalty is a linear function of the number of matrix elements. If we focus on the cost for marshalling integer matrices, the penalty incurred for transposing the matrix has the same order of magnitude as that for swapping bytes. Without an accurate measure it is impossible to tell which context has the lower cost for marshalling integer matrices.

A second source of uncertainty is due to compiler optimization. Good programming practice tells us that the marshalling routines should be modular. In a system like Argus, the programmer provides marshalling routines for `matrix_2d` that handle matrices of any element type [HL82]. The compiler provides the marshalling code for integers and floats, and the linker puts it all together. A good compiler will perform some optimizations such as in-lining to eliminate procedure call overhead (especially in loops), loop unrolling, strength reduction and code movement. It is highly likely that the result of specific optimizations will depend on the marshalling routine supplied for a particular element. Furthermore, the result is not easy to predict. This means that we cannot have a quantitative, modular cost model that takes a parameterized cost equation for `matrix_2d` and plugs in the (separately computed) cost for a specific element.

The final source of uncertainty is due to vspaces that share components in a context and how this interacts with the statistical behavior of the peer. For example,

because matrix_2d[**int32**] and matrix_2d[**float32**] share the `matrix_2d` component, there are assignments that are not optimal for both vspaces. Consider the two assignments

{`matrix_2d:rowmaj`, `int32:le`, `float32:ieee`} and
{`matrix_2d:rowmaj`, `int32:be`, `float32:vax`}.

The first is optimal for matrix_2d[**int32**] but suboptimal for matrix_2d[**float32**], whereas the opposite is true for the second. In comparing these two assignments, the one with the least cost depends on several statistics. The first is the relative sizes of the two matrices. If the integer matrices are sufficiently larger on average than the floating-point matrices then the first context has a lower cost, and vice versa. The second is the relative frequency of different messages. If integer matrices are sent in different messages than floating-point matrices then the relative cost of the two contexts depends on the ratio of messages sent.

A third factor, which is not illustrated here, involves tagged unions (i.e., `oneof`). Because different tags may have different vspaces, a situation analogous to that above occurs where a context is not optimal for all the vspaces. In this case the comparison of contexts depends on an accurate characterization of the distribution of tag values. Without an accurate knowledge of a peer's statistical behavior it is impossible to compare the costs of contexts that are not optimal for all vspaces.

In summary, the lack of accuracy introduced by programmer-supplied code together with the uncertainty of program statistics and compiler optimizations make it very difficult to devise a quantitative cost model. The lack of accuracy can only be addressed by providing the programmer with tools for measuring the costs of compiled code. Better program statistics can only be captured by measuring actual operation. The amount of work required to provide accurate costs in both cases is quite high. As the next section shows, there are many cases where it is not necessary to have accurate costs in order to determine that one context is better than another. A qualitative analysis can provide much information with significantly less work.

3.2.2 Context Lists

It is often the case that a programmer can state that one context has a lower cost than another, even though the exact magnitude of the difference is not known. The following examples demonstrate this.

First, consider the following two assignments for the example in the preceding section:

{`matrix_2d:rowmaj`, `int32:le`, `float32:vax`} and
{`matrix_2d:rowmaj`, `int32:be`, `float32:vax`}

It is clear that the first context has a lower cost than the second because the only difference between them is the representation for `int32`, which is optimal in the first context.

As another example consider:

{`matrix_2d:colmaj`, `int32:le`, `float32:vax`} and
{`matrix_2d:colmaj`, `int32:be`, `float32:ieee`}

Again, the first has a lower cost than the second because the cost differences for both `int32` and `float32` favor the first.

The pair of assignments in the second example above illustrates one method for constructing preferences. The argument above was based on looking at the costs for matrix_2d[**int32**] and matrix_2d[**float32**] separately and noticing that the cost differences contributed by the vspaces were combined in a consistent manner, i.e., both favored the first context over the second. In general, our method will be to divide the negotiable set into subsets, analyze the individual costs of each subset and then combine them. This idea is pursued further in Chapter 4.

All the example cases above illustrate differences that result from comparing two representations for the same vspace component. The following example shows that it is possible to compare the costs of two different vspaces. Consider a negotiable set in which integers and floats occur a fixed number of times, e.g., as single arguments or results, or as fields of a record. Furthermore, the peer runs on a Vax and the programmer knows that the cost of converting floats is several times higher than that for converting integers. In this case the context {`float32:vax`, `int32:be`}

is preferred to {`float32:ieee`, `int32:le`} because the cost of marshalling floats dominates the total cost.

As yet another case, a programmer may wish to tune a given application by instrumenting the program to measure the actual costs of using certain commonly used contexts during operation. This approach is viable because only a few of the many possible contexts will actually be measured. Using this information the programmer may be able to state preferences among these contexts.

Finally, the cost differences between contexts may be unimportant because the cost of marshalling is small compared to the cost of sending messages. This is the case when very little data is sent or many blocking remote procedure calls are made without sending or receiving much data. In such cases just about every non-optimal context is equally preferred, and it is not worth the effort to order them.

What all the examples above have in common is that some contexts can be ordered without a lot of analysis while for others the order is either unknown, uncertain or unimportant. In other words, the relationship among the contexts is that of a partial order. This observation leads to the following definition.

A *context list* for a negotiable set S is a tuple consisting of a set of contexts for S and a strict partial order on those contexts. The set of contexts of a context list C is called its *domain* and is denoted by $dom(C)$. The relation that orders the domain of C is denoted by $\mathcal{R}_C$. By definition, $is_context(c, S)$ is true for all $c \in dom(C)$. Finally, if x and y are contexts in $dom(C)$ and $\langle x, y \rangle \in \mathcal{R}_C$, then we say that x *is preferred to* y.

Context lists capture exactly the qualitative ordering of contexts as described in the examples in this section. However, as one might imagine, expressing a context list directly in terms of individual contexts could be rather tedious if there are many vspaces in a negotiable set and, as a result, many contexts. The problem of expressing context lists is postponed until Chapter 4 where it receives considerable attention. All that remains at this point is to see how context lists are used in negotiation.

3.3 Preferences and Negotiation

Negotiation is the mechanism by which two peers decide on the best context for a negotiable set. During negotiation the peers exchange context lists. The details of the protocol for exchanging context lists are not considered here. What matters is that at some point at least one peer has both context lists in its possession and computes the result. The question, then, is what is the best context.

Given that a context list is based on a qualitative cost model for one peer, it is not possible to choose the best context based on comparing contexts from the context lists of two peers. The best we can do is choose the context with lowest cost as specified by one peer. In order to decide which peer's context list is chosen, every peer is assigned a numerical priority. The priorities are exchanged during negotiation, along with the context lists. If two peers have the same priority then one is chosen arbitrarily. Intuitively, the best context is the one that is common to both peers and that is most preferred by the peer with higher priority. Because context lists are partial orders there may be more than one context that meets the criteria. In this case the context list of the lower priority peer is used to choose among them.

In order to make the definition of "best" formal we first introduce the function $max_contexts(C, S)$, which computes the set of maximal contexts in context list C that are also in the set of contexts S:

$$max_contexts(C, S) = \{c \mid (c \in (dom(C) \cap S)) \wedge \nexists c' \in S \text{ s.t. } \langle c', c\rangle \in \mathcal{R}_C\} \quad (3.5)$$

If C_1 and C_2 are the context lists of the peers with higher and lower priority, respectively, then the set of best contexts is given by the following definition:

$$best_contexts = max_contexts(C_2, max_contexts(C_1, dom(C_2))) \quad (3.6)$$

which finds the maximal contexts with respect to the peer with higher priority, and then uses that result to find the maximal contexts with respect to the peer with lower priority. Note that the final result is a set from which one context is chosen arbitrarily. By the definition of *max_contexts* above, any context in the result is supported by both peers. Finally, the result may also be the empty set, in which

case communication is not possible. Ensuring that any two peers can communicate requires imposing some system-level constraint. One approach is to designate one or more canonical contexts for each negotiable set. By definition, every peer supports the canonical contexts, so the failure to negotiate another context does not affect communication.

At this point it is worth discussing what can be accomplished using the priority mechanism described above. Basically, one can designate certain peers as the ones for which communication must be optimized. One strategy is to assign important servers higher priority so that they will not be overloaded with marshalling data. Another would assign certain machines a higher or lower priority depending on their use. For example, you may not want your Cray to waste time marshalling data. Likewise, you may not want your PCs to spend extra time marshalling data, thereby reducing their response time.

With the proper negotiation and communication protocols the priority scheme could be expanded to handle receiver priority where the transmitter sends data using the preferred context of the receiver. This approach has been advocated by researchers in the networking community who have observed that the receiver is usually the one that has the most difficulty keeping up with the transmitter [CT90]. Of course, this requires the use of two contexts, one for each direction.

One could even imagine a resource-management layer that dynamically adjusts the priority of peers based on some shared goal. For example, if the goal is to balance the load between two peers, then the resource managers of each peer could exchange load measurements and use that as a basis to determine priority. This may be stretching things a bit far, but it is not inconceivable.

Chapter 4

Expressing Context Lists

This chapter focuses on how a programmer expresses a context list. Recall from the previous chapter that a context list is a partial order on a set of contexts for a single negotiable set. The main problem is that there may be many contexts that an application can support for a single negotiable set. For example, consider a negotiable set with five vspaces, each of which has two representations. There are potentially thirty-two contexts that a program could support. A larger or more complex set could have many more. Although it is conceivable that a programmer might directly specify the relative preference of every pair of contexts, that would be extremely laborious and error prone.

4.1 The Approach

The approach described in this chapter is to divide a negotiable set into subsets, express the preferences for each subset separately and then combine them to form the result. The division is usually hierarchical, i.e., the negotiable set is divided into subsets consisting of vspaces and vspace components, which are then further subdivided. At this point we can no longer use contexts to describe the representations for a subset. This is where assignments, which were first introduced in Section 3.1.2, come to the rescue.

An assignment describes one representation for each vspace in a subset of a negotiable set or set of vspace components. The preferences among the representations

of a subset are described by a partial order on the corresponding assignments. The partial order on the assignments is called a preference list. The subsets, along with their preferences, are then combined using operations on preference lists that are based on the following methods of division:

1. Identify a subset of vspaces or vspace components whose cost dominates the total cost. For example, if the negotiable set contained only 32-bit integers and 32-bit floating point numbers, one could express the preferences for each separately. The expression combining them would state that the cost of converting floating point numbers is much greater than that of integers.

2. Divide a set of vspaces or vspace components into subsets that can be described independently, but whose combination involves uncertain factors. For example, the relative cost contributions of 32-bit integer matrices and 32-bit floating-point matrices depends on their relative sizes and frequency of use, neither of which may be known. In this case the uncertainty must be reflected in the resulting combination.

3. Divide a set of assignments into subsets, describe the preferences for each subset and then take the union of the individual preferences. This method is useful when there are representations for which there is optimized marshalling code or when the number of assignments is small enough to divide directly.

Informally, a preference-list expression denotes a partial order on a set of assignments. However, there is a fair amount of material required to make this definition precise as well as to define the operations on preference lists.

The remainder of this chapter defines preference lists and the operations that combine them. It is organized as follows. First, some additional material about assignments is introduced to describe their operations. Next, the definition of a preference list is given and the relationship of preference lists to context lists is discussed. The preference-list operators are then introduced with a discussion of how they are used in the decomposition of a negotiable set or vspace. This is followed by a precise definition of each operator.

4.2 More on Assignments

As defined in Section 3.1.2, an assignment is a partial function from vspace components to representations that maps a given component to a representation for that component. The remainder of this section introduces additional material on assignments that is needed to define preference lists and the operations on preference lists. In particular it defines operations on assignments that allow the programmer to construct "larger" assignments from "smaller" ones.

The *restriction* of an assignment a to a set of vspace components U is written as $a \mid U$ and is defined as follows:

$$a \mid U = a' \text{ where } sig(a') = (U \cap sig(a)) \wedge \forall u \in (U \cap sig(a)) \,.\, a'(u) = a(u) \quad (4.1)$$

Restriction is used to construct an assignment whose signature is a subset of the original assignment and whose mapping for that subset is identical. For example, {`matrix_2d:rowmaj`, `int32:be`} | {int32} = {`int32:be`}.

Two assignments can be *joined* to form a new assignment by the operator $\|$, which is defined as follows:

$$a_1 \| a_2 = x \text{ where } sig(x) = (sig(a_1) \cup sig(a_2)) \wedge (x \mid sig(a_1) = a_1) \wedge (x \mid sig(a_2) = a_2) \quad (4.2)$$

if x does not exist then $a_1 \| a_2$ is undefined.

Note that if x does not exist then it must be the case that there is some $u \in (sig(a_1) \cap sig(a_2))$ such that $a_1(u) \neq a_2(u)$. Typically, the signature of the resulting assignment is a superset of the signatures of the two original assignments. In this sense the join operator usually constructs a larger assignment from two smaller assignments. For example, {`int32:be`} $\|$ {`float32:ieee`} = {`int32:be`, `float32:ieee`}.

The join operation can be extended to sets of assignments. If A_1 and A_2 are sets of assignments then their join is defined by:

$$A_1 \| A_2 = \{a_1 \| a_2 \mid a_1 \in A_1 \wedge a_2 \in A_2\} \quad (4.3)$$

4.3 Preference Lists

Intuitively, a preference list is a partial order over a set of assignments. However, not every set of assignments is suitable for a preference list. For example, it does not make sense to compare the assignments {`int32:be`} and {`float32:ieee`} because they have no vspace components in common. Comparing them is a bit like comparing apples and oranges. These two assignments cannot be used to describe preferences for the same subset of any negotiable set. Therefore, before giving a formal definition for preference lists we first define which sets of assignments are suitable. This in turn requires us to define when it makes sense to compare two assignments.

Intuitively, two distinct assignments are said to be "comparable" if it makes sense to compare them. A more direct statement is somewhat difficult and is best illustrated by examples. The example in the previous paragraph in which the assignments have no vspace component in common illustrates one case of incomparable assignments. But it is not sufficient for two assignments to have a set of vspace components in common to be comparable. As an example, consider the assignments {`int16:be`, `int32:be`} and {`int16:be`, `float32:ieee`}. Although they have a vspace component in common, they assign the same representation to that component. The remaining components are incomparable, just as they were above. In order to be comparable, the assignments must specify different representations for at least one common vspace component.

In addition, the definition of comparable must not be too restrictive. In particular, it must not require the assignments to have the same signature. For example, the CIF form of a VLSI mask that is defined in Chapter 5 is an abstract vspace with a binary and a textual representation with corresponding assignments of the form {`cif:bin`, `int32:le`} and {`cif:text`, `char:ascii`}. It makes sense to compare these assignments because the dissimilar components depend on the mapping for `cif` and are necessary for a complete description of the representation.

As shown by the above examples, it is difficult to formalize completely what it means for two assignments to be comparable. I have chosen to make the definition as liberal as possible, imposing only the minimum restrictions that are needed to

simplify the preference-list operations defined in Section 4.3.2. These considerations result in the following definition:

$$comparable(x, y) \Leftrightarrow \exists\, u \in (sig(x) \cap sig(y)) \text{ s.t. } x(u) \neq y(u) \tag{4.4}$$

The above says that x and y are comparable if and only if they have at least one vspace component in common, and they map at least one of those common components to different representations. As a corollary, it implies that neither x nor y is a degenerate assignment, i.e, an assignment whose signature is the empty set.

If A is a set of assignments and if every pair of assignments in A is comparable then we say that A is *coherent.* We are now ready to give a formal definition for a preference list.

A *preference list* is a tuple consisting of a coherent set of assignments and a strict partial order on those assignments. The set of assignments of a preference list P is called its *domain* and is denoted by $dom(P)$. The relation that orders the domain of P is denoted by $\mathcal{R}_P$. If x and y are assignments in $dom(P)$ and $\langle x, y\rangle \in \mathcal{R}_P$ then we say that x *is preferred to* y.

The relationship of preference lists to context lists is quite simple. Every context list is also a preference list. That is because every context is an assignment and the definition of a context as given in Section 3.1.3 ensures that any two contexts for the same negotiable set are comparable. The converse is not true, however. A preference list P is a context list for a negotiable set S only if every assignment in P is a context for S.

4.3.1 Preference-List Operators

The basic approach to constructing a context list for a negotiable set is to divide it into subsets, construct preference lists for each subset and then combine the resulting preference lists. The same approach is used to construct a preference list for a vspace that consists of several components.

There are essentially two methods for dividing up a negotiable set or vspace. The first divides the set of vspace components into subsets. For a negotiable set the division is usually along the lines of individual vspaces. For example, for the

negotiable set from Section 3.2 we would divide it into the vspaces matrix_2d[**int32**] and matrix_2d[**float32**]. For a single vspace like a struct, we could divide it into subsets corresponding to the vspace of each field. For a vspace like an integer matrix we could divide it into integer and matrix components.

Using this method of division results in several preference lists where each assignment in a given list has a signature that is a subset of the vspace components in the original negotiable set or vspace. When combining such preference lists the assignments in each are joined to form larger assignments. The operators that combine preference lists in this manner are called *join operators* because the assignments in the operands' preference lists are joined together to form the assignments in the resulting preference list.

The second method divides a set of assignments into subsets that can be ordered separately. This method is used when the number of assignments is small, such as when the negotiable set has been divided using the first method until the individual preference lists are small enough to express directly in terms of the individual assignments. For example, in the case of integer matrices there may be several marshalling routines, some of which are optimized for a given representation. In this case we could divide the assignments into subsets corresponding to optimized and unoptimized assignments and describe each with a separate preference list. They would then be combined by unioning the subsets along with their preferences. The operators that combine preference lists in this manner are called *union operators* because the assignments in the operands' preference lists are unioned together to form the assignments in the resulting preference list.

The following description of the preference-list operators begins with the union operators because they usually form the basis on which larger preference lists are built. They are good for directly expressing preferences among a small set of assignments and are also the easiest to understand.

4.3.2 Union Operators

There are two union operators: dominant union and conjunction. Dominant union is used to express that one set of assignments is preferred to another. In this case

the first set dominates the second. Conjunction is used to express the merging of preferences from two preference lists that do not specify conflicting preferences.

4.3.2.1 Dominant Union

The binary operator symbol ">" denotes the *dominant union* of its arguments. It is a partial function and is defined if and only if the arguments' domains are disjoint and their union is coherent. More precisely, if P_1 and P_2 are preference lists then $P_1 > P_2$ is defined if and only if $dom(P_1) \cap dom(P_2) = \emptyset$ and $coherent(dom(P_1) \cup dom(P_2))$. Given these restrictions the dominant union is defined as follows:

$$dom(P_1 > P_2) = dom(P_1) \cup dom(P_2) \tag{4.5}$$

$$\langle x, y \rangle \in \mathcal{R}_{(P_1 > P_2)} \Leftrightarrow (x \in dom(P_1) \wedge y \in dom(P_2)) \vee \langle x, y \rangle \in \mathcal{R}_{P_1} \vee \langle x, y \rangle \in \mathcal{R}_{P_2} \tag{4.6}$$

From the definition it is easy to see that in the result, every assignment from P_1 is preferred to every assignment from P_2. In addition, the ordering on assignments from the same preference list is preserved. It is also clear why the arguments' domains must be disjoint. Otherwise, for all $x \in (dom(P_1) \cap dom(P_2))$ we would have that $\langle x, x \rangle \in \mathcal{R}_{(P_1 > P_2)}$, which is clearly nonsense.

An example use of > is given by the expression `{int32:be} > {int:le}` which says that the big-endian representation for 32-bit integers is preferred to little-endian.

The operator > is associative, so parentheses can be omitted when writing expressions. A more interesting example using > will have to wait until after the definition of the conjunction operator.

4.3.2.2 Conjunction

The binary operator symbol "&" denotes the *conjunction* of its arguments. It is a partial function and is defined if and only if both the union of the arguments' domains is coherent and the union of the arguments' relations is consistent. More precisely, let $P = (dom(P_1) \cup dom(P_2))$ and $\mathcal{R} = transitive_closure(\mathcal{R}_{P_1} \cup \mathcal{R}_{P_2})$. Then $P_1 \,\&\, P_2$ is defined if and only if P is coherent and for all $a \in P$, $\langle a, a \rangle \notin \mathcal{R}$. The latter constraint states that the transitive closure of the union of the arguments' partial

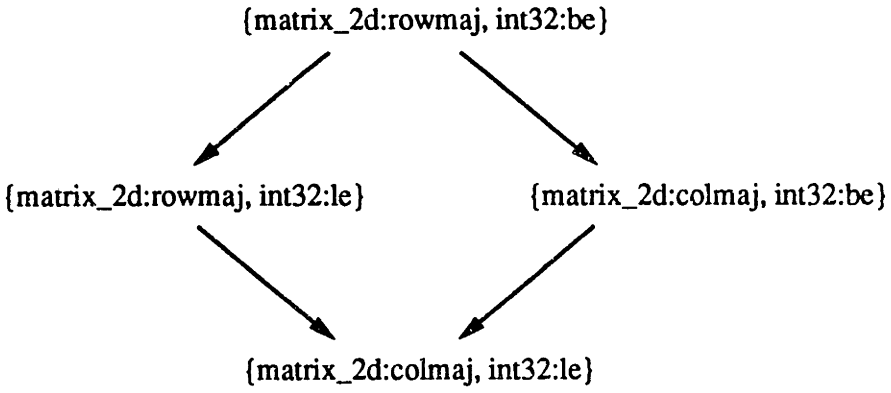


Figure 4-1: Result of Conjunction

orders must not contain any cycles. Cycles correspond to inconsistent information. Given these restrictions, conjunction is defined by:

$$dom(P_1 \,\&\, P_2) = dom(P_1) \cup dom(P_2) \tag{4.7}$$

$$\mathcal{R}_{(P_1 \& P_2)} = transitive_closure(\mathcal{R}_{P_1} \cup \mathcal{R}_{P_2}) \tag{4.8}$$

Note that & is both associative and commutative, so parentheses may be omitted when writing expressions.

The following example uses both union operators to express the preference list given in Figure 4-1. In the figure, preferences between assignments are indicated by arrows. The preference list corresponds to the example detailed in Section 3.2.1, but restricted to assignments for the vspace matrix_2d[**int32**]. In this case the cost for invoking the programmer-provided procedure for transposing the matrix relative to the cost for swapping bytes is unknown. Therefore, the two corresponding assignments are unordered in the preference list. The following is an expression for this preference list.

```
({matrix_2d:rowmaj, int32:be} > {matrix_2d:rowmaj, int32:le})
& ({matrix_2d:rowmaj, int32:be} > {matrix_2d:colmaj, int32:be})
& ({matrix_2d:rowmaj, int32:le} > {matrix_2d:colmaj, int32:le})
& ({matrix_2d:colmaj, int32:be} > {matrix_2d:colmaj, int32:le}
```

This example can be more concisely expressed as:

```
    {matrix_2d:rowmaj, int32:be}
  >
    ({matrix_2d:rowmaj, int32:le} & {matrix_2d:colmaj, int32:be})
  >
    {matrix_2d:colmaj, int32:le})
```

4.3.3 Join Operators

There are two join operators: the dominant join and the consistent join. For both operators the relationship between any two assignments in the result is a function of the partial order on *all* the assignments from both operands that joined to form those two assignments. In other words, if assignments a_1 and a_2 are in the result then we must consider all the assignments that joined to form them. If looked at in terms of cost this means that the cost associated with each assignment that joined to form the result must be factored into the cost associated with the result. This can be quite complex and difficult to understand if there are several assignments from each operand that joined to form the same assignment in the result.

The complexity of having to consider multiple assignments in each operand is reduced substantially by restricting the domain of a preference list to a coherent set. As shown by the following theorem, for a given assignment in the result there is exactly one assignment from each operand that joined to produce it. Not only does this simplification make the definitions easier to understand, it also makes it easier for the programmer to reason about how the operands combine to give the desired result.

Theorem 1 - If A_1 and A_2 are coherent sets of assignments and $a \in (A_1 \,\|\, A_2)$, then there exists exactly one $a_1 \in A_1$ and exactly one $a_2 \in A_2$ such that $a = a_1 \,\|\, a_2$.

Proof.

First, by the definition of $\|$ on sets of assignments there is at least one assignment $a_1 \in A_1$ and one assignment $a_2 \in A_2$ such that $a = a_1 \,\|\, a_2$. The rest of the proof is by contradiction. Assume that there exists an $a_1' \in A_1$ and an $a_2' \in A_2$ such that $a = (a_1 \| a_2) = (a_1' \| a_2')$. Because the proof is symmetric for a_1 and a_2, we can assume

w.l.o.g. that $a_1 \neq a_1'$. Now, by definition of coherence, a_1 and a_1' are comparable. Therefore, there exists a u such that $a_1(u) \neq a_1'(u)$. By the definition of $\|$ on two assignments, we have that $(a_1 \parallel a_2)(u) \neq (a_1' \parallel a_2')(u)$, which is a contradiction. Note that the proof does not depend on a_2 and a_2' being distinct, so the subcases corresponding to whether or not they are distinct need not be considered separately. Q.E.D.

In addition, it is not necessary for the operators to place restrictions on the operands as was the case with the union operators. This is because the join of two coherent sets is coherent as shown by the following theorem.

Theorem 2 - If A_1 and A_2 are coherent sets of assignments, then $A_1 \parallel A_2$ is also coherent.

Sketch of Proof.
The proof uses Theorem 1. Because any two distinct assignments in the resultant join are produced by exactly two distinct assignments from at least one operand, and these assignments are comparable, then the resulting assignments are also comparable.
Q.E.D.

Note that it is possible for the resulting set to be empty. This occurs when every assignment in one operand disagrees with every assignment in the other operand on the mapping for some common vspace component. When this occurs, it most likely indicates a mistake on the part of the programmer and should result in a warning from the stub compiler.

We are now ready to give definitions for each operator.

4.3.3.1 Dominant Join

The binary operator symbol "$\gg$" denotes the dominant join of its arguments. The domain of the result is given by the following:

$$dom(P_1 \gg P_2) = dom(P_1) \parallel dom(P_2) \tag{4.9}$$

Let x and y be elements of $dom(P_1 \gg P_2)$. By Theorem 1, there exists exactly one $x_1 \in P_1$, $x_2 \in P_2$, $y_1 \in P_1$, and $y_2 \in P_2$ such that $x = x_1 \parallel x_2$ and $y = y_1 \parallel y_2$. The

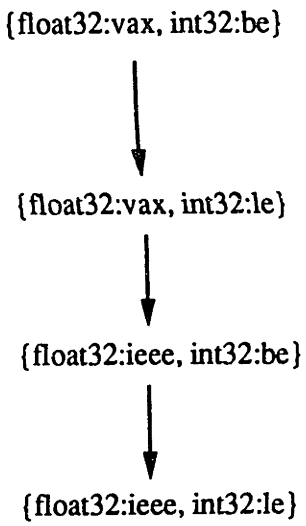


Figure 4-2: Dominant Join of Simple Preferences

definition for the partial order of the result is as follows:

$$\langle x, y \rangle \in \mathcal{R}_{(P_1 \gg P_2)} \Leftrightarrow \langle x_1, y_1 \rangle \in \mathcal{R}_{P_1} \vee (x_1 = y_1 \wedge \langle x_2, y_2 \rangle \in \mathcal{R}_{P_2}) \tag{4.10}$$

Intuitively, the left-hand-side operand dominates in that any preference in P_1 is always present in the result, regardless of any preferences in P_2. Preferences in P_2 are present only when they result from joining assignments in P_2 with the same assignment in P_1. In terms of cost, this means that the cost difference between any two assignments in P_1 is significantly greater than that between any two assignments in P_2. Therefore, the cost difference in the result is dominated by the contribution of the assignments from P_1.

The dominant join is most useful for expressing that one set of vspace components is more important. A simple example is given by the following expression:

$$(\{\texttt{float32:vax}\} > \{\texttt{float32:ieee}\}) \gg (\{\texttt{int32:be}\} > \{\texttt{int32:le}\})$$

The resulting preference list consists of four assignments corresponding to the two choices for each component and is shown in Figure 4-2. Because floating-point conversion is significantly more expensive than integer conversion, the preference list for `float32` dominates.

4.3.3.2 Consistent Join

The binary symbol "$\|$" denotes the *consistent join* of its arguments. The domain of the result is given by the following:

$$dom(P_1 \parallel P_2) = dom(P_1) \parallel dom(P_2) \tag{4.11}$$

Let x and y be elements of $dom(P_1 \parallel P_2)$. By Theorem 1, there exists exactly one $x_1 \in P_1$, $x_2 \in P_2$, $y_1 \in P_1$ and $y_2 \in P_2$ such that $x = x_1 \parallel x_2$ and $y = y_1 \parallel y_2$. The definition for the partial order of the result is as follows:

$$\begin{aligned} \langle x, y \rangle \in \mathcal{R}_{(P_1 \| P_2)} \Leftrightarrow \; & (\langle x_1, y_1 \rangle \in \mathcal{R}_{P_1} \wedge \langle x_2, y_2 \rangle \in \mathcal{R}_{P_2}) \\ & \vee (x_1 = y_1 \wedge \langle x_2, y_2 \rangle \in \mathcal{R}_{P_2}) \\ & \vee (\langle x_1, y_1 \rangle \in \mathcal{R}_{P_1} \wedge x_2 = y_2) \end{aligned} \tag{4.12}$$

As can be seen, two assignments in the result are ordered if and only if 1) the assignments from both operands that joined to form them are ordered in the same way, or 2) they resulted from the join of two ordered assignments from one operand with the same assignment from the other operand. The operator is called consistent because the partial order of the result is consistent with the order of *both* operands.

As a simple example consider the following expression

$$(\{\texttt{matrix_2d:rowmaj}\} > \{\texttt{matrix_2d:colmaj}\}) \parallel (\{\texttt{int32:be}\} > \{\texttt{int32:le}\})$$

which also denotes the preference list in Figure 4-1.

Another important aspect of this operator is that it preserves certainty. Two assignments are ordered only when we are certain that one has a higher cost than the other. Similarly, they are unordered when the cost difference between them is unknown or uncertain. In other words, the cost difference in the result is certain (ordered) only if the cost difference of the operands is certain and both are ordered the in the same manner. If they are ordered differently then the result remains unordered because there is no information about the relative contribution of each operand.

Therefore, the consistent join operator is used when we are uncertain about the cost contributed by each operand to the result. For example, in the case of integer and floating-point matrices, if we have no information about the relative

sizes of integer-matrix values and floating-point-matrix values, then we must use the consistent join to combine them.

Chapter 5

Abstract Vspace Examples

This chapter gives examples to illustrate the use of abstract vspaces with multiple representations and the construction of preference lists. The examples examined here are a video display image abstraction, which is discussed in Section 5.1, and a VLSI design abstraction, which is discussed in Section 5.2. Each section describes the abstraction, the representations, and a preference list.

The examples focus on individual vspaces. The approach for constructing preference lists for individual vspaces is similar to that for interfaces, namely, divide and combine. The representative preference list described in each section is based on the cost of marshalling data between a single internal program representation and each of the external representations.

5.1 Video Display Image

A video display image consists of a rectangular plane of colored pixels. Each pixel, when projected on a display device, generates a single point of the image. An image has a height and a width, both of which are specified by the number of pixels in the respective dimension. The density of the pixels in a given image is referred to as the image's resolution and is specified in pixels per linear distance (e.g., "dots" per inch). The resolution information allows an image that was captured on one device to be displayed on another device with a different resolution. In order to keep the example simple, resolution information is not considered in the remainder of this

section.

A pixel consists of three components, one for each axis of a three-dimensional color space. The color model is based on the NTSC RGB (red, green, blue) color model, which specifies color in terms of an RGB video camera signal as used in broadcast television [NH88]. In the RGB system a pixel's value is obtained by sampling the light spectrum of the pixel's color at three standard frequencies corresponding to red, green and blue. The samples are then normalized with respect to a standard reference value for white and quantized to one of 256 levels, giving eight bits per sample. The three samples together form an RGB triple that specifies the pixel's value.

Although the basic abstraction described above is simple, the representations are actually quite complex. The complexity arises from the need to reduce storage and transmission bandwidth requirements, and to accommodate various recording and display devices. Because the vspace representations are complex, their description is given in two parts. The first part describes the concepts underlying the representations, and the second maps these concepts into specific vspace representations.

The representations described below are based on a subset of the proposed JPEG standard for the compression of continuous-tone still images [ISO91]. A good overview of JPEG is given in [Wal91]. The standard actually describes a series of representations based on four modes of operation and several parameters. The subset considered here is based on the basic, discrete cosine transform (DCT) mode with a color space consisting of three components of eight-bit samples each. The DCT mode involves compression which, for the purpose of this example, is modeled simply as a function that is applied to the data stream.

The two basic concepts underlying the representation of a video image are the color model and the organization. The color model specifies how individual samples are combined to form a pixel's color. The organization specifies how samples are blocked into groups for compression and the order in which blocks are transmitted. The color model and organization are described in the next two sections. The vspace definitions that represent these concepts are given in Section 5.1.3, followed by an example preference list in Section 5.1.4.

5.1.1 The Color Model

The description of the underlying concepts begins with the color model. There are actually two parts to the color model. The first is the coordinate system that is used to specify color. The two coordinate systems described in this example are based on combining light of three different spectra to form a color. The three spectra constitute a three-dimensional coordinate system. Each color, then, has three components corresponding to the contribution of each spectrum. The second part of the model is the relative spatial resolution of the individual components. As will be seen, the relative spatial resolution is not independent of the coordinate system.

The first color coordinate system is NTSC RGB, which samples a color at three standard frequencies corresponding to red, green and blue and then normalizes the sample with respect to a reference white. The resultant sample values all fall in the range 0.0-1.0 and are quantized to 256 levels. The normalization allows the original color to be faithfully reproduced on diverse television receivers.

The second coordinate system is YUV, which is a linear transformation of RGB that separates the three components into one luminance (i.e., brightness) component and two chrominance (i.e., color) components. The transform is given by the following equations:

$$\begin{aligned} Y &= .299R + .587G + .114B \\ U &= (B - Y)/2.03 \\ V &= (R - Y)/1.14 \end{aligned} \tag{5.1}$$

Note that the U and V values may be negative. In order to be faithful to the abstraction, we must restrict the set of YUV triples to those values that correspond to valid RGB triples. This restriction is not a problem in practice because the YUV triples are usually generated from RGB triples.

The reason for using the YUV representation is that the eye is more sensitive to brightness than to color with respect to spatial resolution. We can take advantage of this to reduce the number of samples needed to represent an image's color components. This brings us to the spatial aspect of the color space, which involves the ratio of the number of samples in the luminance component to the number in the chrominance components.

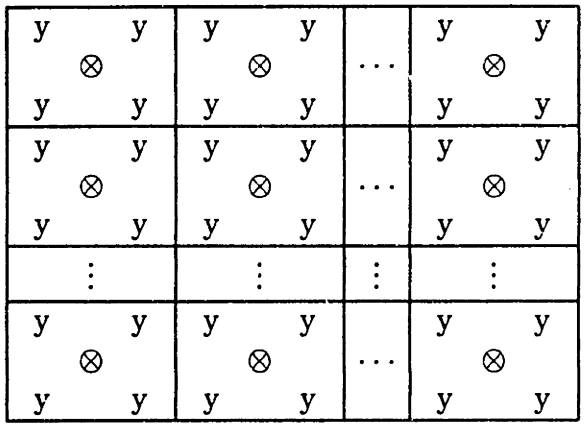


Figure 5-1: Spatial Relation of Components, 4:1:1 Ratio

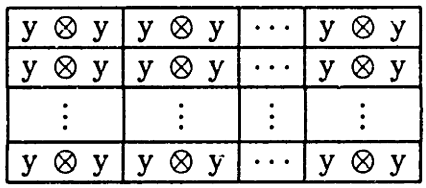
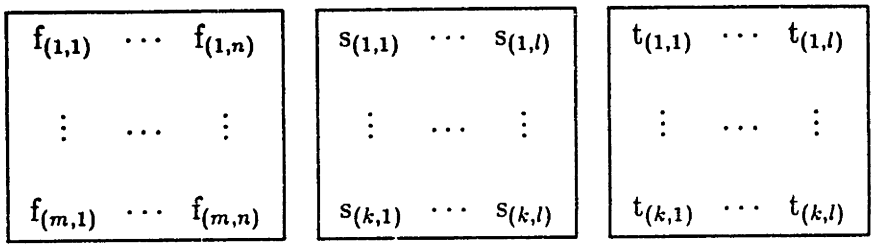


Figure 5-2: Spatial Relation of Components, 4:2:2 Ratio

One common ratio for the number of samples in the YUV components is 4:1:1, which corresponds to the luminance component having twice the vertical and horizontal resolution as the chrominance components. In this case there are four times as many Y samples as U and V samples, hence 4:1:1. The spatial relationship of the component samples is illustrated in Figure 5-1. In the figure, each "y" corresponds to a luminance sample and each "⊗" to both chrominance samples (i.e. one U and one V sample). Each chrominance sample covers an area that is four times greater than that of a luminance sample.

Another commonly used ratio is 4:2:2 where the horizontal resolution of the luminance component is twice that of the chrominance components, and the vertical resolution is the same. The spatial relationship among component samples in this case is illustrated in Figure 5-2.

With respect to the abstract vspace, each "y" sample in Figures 5-1 and 5-2 corresponds to one pixel in the image. To obtain the abstract RGB triple for a pixel, the U and V components are interpolated and the resulting YUV triple converted to an RGB triple. In the 4:1:1 ratio the samples are interpolated both horizontally and vertically, whereas in the 4:2:2 ratio they are interpolated only horizontally.



Component 1 Component 2 Component 3

Axes (fst)	Ratio	
RGB	$m = k$	$n = l$
YUV	$m = k$	$n = l$
YUV:4:1:1	$m = 2k$	$n = 2l$
YUV:4:2:2	$m = k$	$n = 2l$

Figure 5-3: Component Structure

The ability to use different spatial resolutions for the components results from the separation of the luminance and chrominance information in the image. Therefore, the sample ratio is not independent of the choice of coordinate axes for the color space. For example, in the case of RGB, the ratios must all be one because every component carries color information. In the case of YUV, the U and V components must have the same resolution.

So far the color model has been described as if all the component samples were overlaid. The reason for this is to illustrate the spatial relationship of the components. However, in the actual representation, the components are separated into three planes, where each plane contains the samples of one component. This is illustrated in Figure 5-3, where "f", "s" and "t" represent the first, second, and third coordinate axes, respectively, and are either RGB or YUV. The subscripts give the position of each sample in its component plane, and the accompanying table shows the linkage between coordinate axes and sample ratios.

As shown in the table of Figure 5-3 there are four different representations. The RGB, YUV:4:1:1, and YUV:4:2:2 representations have already been described. The YUV representation is included for completeness and because it is usually an intermediate step in the conversion between the RGB and YUV:4:n:n formats.

In summary, there are four representations for the color aspect of an image. The representations involve different coordinate axes for describing the components of the color space and different spatial resolutions for the sampling of each component.

In theory, we may convert an image from one representation to another. In practice there are several issues that must be addressed with respect to converting between them. These are discussed in appendix B.

This completes the description of the color model. We are now ready to see how the individual components are organized for compression and transmission.

5.1.2 Organization of Components

As described above and illustrated in Figure 5-3, the basic organization of the image is by component planes. This section introduces the concepts of blocking and interleaving that impart additional organization to the image's structure. Blocking is needed for compression because groups of samples are compressed as a single unit. Interleaving, or the lack thereof, is needed to accommodate recording and display devices having different needs with respect to buffering and signal generation. Devices can buffer an entire image, generate it incrementally, use separate signals for each component or require a composite signal. The discussion begins with blocking, then covers interleaving and finishes with a very brief description of compression.

Each component is divided into data units where a data unit consists of an eight-by-eight block of samples. There are sixty-four samples per data unit. If a component is not a multiple of eight samples in height or width then it can be padded, the details of which are not considered here. Instead, we assume that images are a multiple of eight samples in each dimension.

The division of components in Figure 5-3 into data units is shown in Figure 5-4. Each upper-case letter F, S and T in Figure 5-4 consists of a block of samples of the corresponding lower-case letter in Figure 5-3. Furthermore, $M = m/8$, $N = n/8$, $K = k/8$ and $L = l/8$. All the samples in a single data unit are compressed together. As described below, compression depends on the order in which the data units are transmitted, which is the next topic.

When an image is transmitted, the components can be sent separately or interleaved. When sent separately, all the data units of one component are sent before those of the next component. Within a component, the units are sent in raster scan order, i.e., units are sent left-to-right witin a row, and rows are sent top-to-bottom.

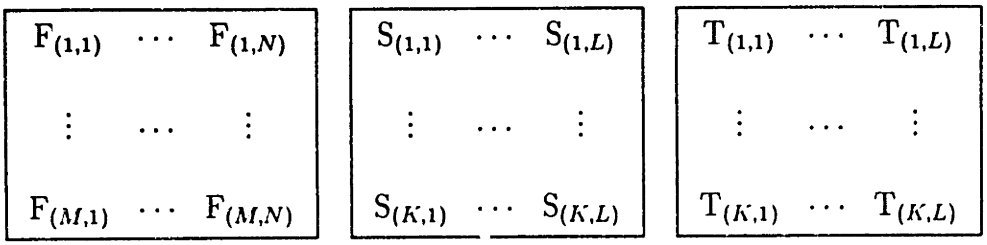


Figure 5-4: Component Division

Interleaving is somewhat more complex because the exact order in which the data units are sent depends on the ratio of the number of samples in the components. The basic idea is that an equal size area from each component is sent. The size of the area is determined by finding the smallest area that holds an integral number of data units from each component. This minimum area is referred to as a *macro-unit.* In the case of RGB, a macro-unit covers one data unit from each component because the components have equal resolution. For YUV:4:1:1, a macro-unit covers a 2x2 square of data units of Y and one data unit of U and V, and for YUV:4:2:2 it covers a 1x2 rectangle of data units of Y and one data unit of U and V. When viewed in terms of macro-units, each component consists of a rectangular array of the same number of macro-units.

When transmitted using interleaving, one macro-unit is sent from each component in order. Within a component the macro-units are sent in raster scan order. If a macro-unit of a componcnt contains more than one data unit, then the data units are transmitted in raster scan order.

Each component is compressed separately, regardless of whether the data units are transmitted separately or interleaved. Furthermore, compression is performed one data unit at a time and in the order in which the component's units are transmitted. Compression uses the Discrete Cosine Transform, which is related to the two-dimensional Fast Fourier Transform. The transformed data is further compressed by quantization, run-length encoding, and Huffman coding. The exact details of compression are not that important for understanding the example. Therefore, compression is modeled as a function that is applied to the data units of each component separately.

```
video-display-image = abstract
      rep rgb = component-order[dct-unit,1,     %red component
                                dct-unit,1,     %green component
                                dct-unit,1]     %blue component

      rep yuv = component-order[dct-unit,1,     %Y component
                                dct-unit,1,     %U component
                                dct-unit,1]     %V component

      rep yuv411 = component-order[dct-unit,4, %Y component
                                   dct-unit,1, %U component
                                   dct-unit,1] %V component

      rep yuv422 = component-order[dct-unit,2, %Y component
                                   dct-unit,1, %U component
                                   dct-unit,1] %V component
      end
dct-unit = unspecified % compressed data unit
```

Figure 5-5: Vspace definition for `video-display-image`.

In summary, components are divided into data units of sixty-four samples each. Data units, in turn, are organized into macro-units corresponding to an equal size area for each component. The macro-units are transmitted in raster scan order, and the components are sent either separately or interleaved. Finally, each component is compressed separately and the data units within a component are compressed individually and in the order of transmission.

5.1.3 Mapping Concepts Into Vspace Descriptions

The mapping of concepts from the previous section into vspaces is rather straightforward. As described above, the representation of an image has two degrees of freedom corresponding to the color model and the organization. The color model is embedded directly into the representations of the abstract vspace `video-display-image`, whose definition is given in Figure 5-5.

An informal specification consists of the opening paragraphs of Section 5.1 on page 79. The abstraction functions and representation invariants are omitted for

```
component-order[comp1:vspace, n1:int,
                comp2:vspace, n2:int,
                comp3:vspace, n3:int]
  = abstract

    rep separate =  struct[first:sequence32[comp1]
                           secnd:sequence32[comp2],
                           third:sequence32[comp3]]

    rep interleaved =
          sequence[
              struct[first:fixed-sequence[n1,comp1],
                     secnd:fixed-sequence[n2,comp2],
                     third:fixed-sequence[n3,comp3]]]
    end
```

Figure 5-6: Vspace definition for component-order.

brevity. Including them would introduce a large amount of detail that is unnecessary given the detailed description already given.

A dct-unit corresponds to a data unit of 8x8 samples that has been compressed. It has one representation whose details are omitted here. For the purpose of the example it can be viewed as a variable length bit vector.

The organization is abstracted into a separate, abstract constructor called a component-order, whose definition is shown in Figure 5-6. The two representations shown in the figure correspond to sending the three components separately and interleaved. When interleaving is chosen, the integer parameters specify how many data units are in a macro-unit for each component. These parameters are ignored when components are sent separately.

The reason for making component-order an abstract constructor is that JPEG has other modes, where individual samples are sent instead of data units. Because it is a constructor, component-order can be reused in these cases to handle the ordering of components.

5.1.4 Preference List

The preference list depends on the internal representation used by a program and the marshalling costs for converting the internal representation to each of the external representations. In this example, the internal representation uses the RGB color space and stores each component separately in raster-scan order. Thus, there are three arrays of bytes, one each for the R, G and B components, where each byte represents one sample. The arrays are ordered by rows, and four bytes are packed into each thirty-two bit word. This representation corresponds to a typical frame buffer found on a small desk-top computer with a bit-mapped graphics display for output or an attached image scanner for input.

Assuming that our computer has enough primary memory to buffer the entire image, the primary marshalling costs for both encoding and decoding are due to:

1. Compression.
2. RGB–YUV conversion.
3. Filtering/interpolation (i.e., YUV–YUV:4:n:n).
4. Message overhead.

Of the above, compression is the most expensive because it requires several hundred CPU cycles per sample. Therefore, representations that send fewer samples, like YUV:4:n:n, are preferred. In comparison, RGB–YUV conversion requires at most a few tens of CPU cycles per sample. The cost of filtering/interpolation is on the order of a few CPU cycles per sample. The effect of filtering/interpolation is to reduce the number of samples significantly. Therefore, its cost can be ignored because it is more than compensated for by the reduction in compression costs. Message overhead is similar to compression in that it favors smaller representations. As a result, message overhead can be correlated with the cost of compression. Because the effects of filtering/interpolation and message overhead can be incorporated into compression, the only factors that determine the context list are compression and RGB–YUV conversion.

At first glance, the dominance of compression and the cost of converting RGB to YUV suggests a context list of the form:

$$(\{\texttt{image:yuv411}\} > \{\texttt{image:yuv422}\} > \{\texttt{image:rgb}\} > \{\texttt{image:yuv}\}) \gg \mathcal{E} \quad (5.2)$$

where $\mathcal{E}$ is a preference list expression for the component ordering and `image` is an abbreviation for `video-display-image`. However, this expression cannot be used because the factors for component ordering are not the same for RGB and any YUV form.

For RGB, sending the components separately or interleaved costs the same because, in both cases, the basic operation consists of extracting a data unit from a component and compressing it. The order of access is irrelevant and, therefore, neither order is preferred.

When any YUV form is used, the order of component transmission becomes important because computing any Y, U, or V sample requires access to all three of the RGB components simultaneously. The same is true on reception with the roles of YUV and RGB reversed.

When the components are interleaved, the corresponding data units from each component are fetched together for transmission. Then the 8-bit samples are unpacked, converted, filtered (if YUV:4:n:n), compressed and the result transmitted. Reception follows the same steps in reverse order, except that the inverse operations are performed (e.g., interpolation instead of filtering). Because all the processing for a given data unit occurs at one time, encoding and decoding require one pass. Furthermore, the transmit/receive buffers can be reused, reducing the cost of buffer management.

In contrast, when the components are transmitted separately, encoding requires at least two passes. In the first pass, all the data units are compressed as above, but the compressed output for the U and V components is buffered until all the compressed Y data units have been sent. The second pass transmits the buffered components. The only alternative to buffering is to make three passes over the RGB image, one for each of the three Y, U and V components.

On reception, the Y and U components must be buffered until the V component starts to arrive, at which point conversion of the data from YUV to RGB can begin.

If the decode procedure is clever, it can use the storage for the final image as a temporary buffer. If this is not possible then the compressed Y and U components are buffered, introducing extra overhead.

Other than buffering, encoding and decoding for separate transmission is similar to that for interleaving. Because of the extra buffering, interleaving is preferred to separate transmission. A correct context list that separates out the contribution of component ordering for RGB and YUV is shown in Figure 5-7. The lines above the outermost & are for the YUV forms and those below for RGB. The subexpression for RGB not only deals with component order, but also places RGB with respect to compression and conversion.

```
let
   order = component-order
in
    (({image:yuv411} > {image:yuv422} > {image:yuv})
      ≫ ({order:interleaved} > {order:separate}))
    &
    (({image:yuv422} > {image:rgb} > {image:yuv})
      ≫ ({order:interleaved} & {order:separate}))
```

Figure 5-7: Context list for video display image

5.2 VLSI Design

The second example of an abstract vspace that I will study in detail is one that describes the design of an nMOS VLSI design. The abstraction is described below. There are actually two conceptual views of the abstraction. The first view is that of the manufacturer, which gives a physical intuition of what a designer must specify. The second is that of the designer, which gives the functional structure of the circuit. The designer must relate the functional structure to the physical structure. The description of the abstraction is followed by Section 5.2.2, which gives three representations for the abstract vspace. This is followed by a discussion of the factors affecting marshalling and a description of the preference list for an example internal representation.

5.2.1 The Abstraction

From the chip manufacturer's point of view, a VLSI design consists of a set of masks, one for each step in the manufacturing process. A mask is basically a stencil that is used to create or modify one layer of a multi-layered physical structure. Each mask is given a name corresponding to the layer it creates or the processing step for which it is used, e.g., polysilicon, diffusion, metal. The description of an individual mask consists of a set of geometric figures corresponding to the holes of the stencil. The figures may abut so that several figures together describe a single hole. In this example, the geometric figures are restricted to rectangles whose edges are either parallel or perpendicular to the edge of the chip. This restriction is referred to as "Manhattan" geometry and is very common in current VLSI design environments. Therefore, from the manufacturer's viewpoint a VLSI design consists of a set of masks, where each mask is a set of rectangles and every rectangle has a position within the mask.

The chip designers who are responsible for creating mask descriptions have a very different view of the design. In their view a VLSI design is a set of hierarchically structured functional units. In addition, a given functional unit can be instantiated any number of times. For example, a full adder consists of two half adders, carry propagation logic and an interconnection network. The half adders and carry logic in turn are constructed from simple, interconnected logic gates. The logic gates are constructed from combinations of physical layers that form transistors, resistors and conductors. Thus, the designer is mostly concerned with the functional structure of the design and is only interested in actual mask geometry at the lowest levels of the design hierarchy.

The basic design abstraction is the *cell*, which is a named, parameterized construct that encapsulates the implementation of a functional unit. A *cell definition* consists of a set of physically located elements, where an element is either an instantiation of (another) cell or a rectangle in a mask. The coordinate system for a cell is a cartesian plane with a horizontal x axis and vertical y axis, where distances along the axes are measured in discrete units. When a cell is designed, the designer chooses an (arbitrary) origin for that cell and specifies the position of each element

with respect to that origin.

A *cell instantiation* consists of a cell reference (using the cell's name) and an ordered list of parameters that specify transformations to apply to the referenced cell. The possible transformations are translation, mirror-x, mirror-y, and rotation. The translation parameter is a vector that is added to the position of each element, thereby translating the entire cell as a single unit to a new position. The mirror parameters specify that the cell should be mirrored (i.e. flipped) about the x or y axis. The rotation parameter must be a multiple of 90 degrees because of the limitation to Manhattan geometry. The parameter list is ordered because the result of applying the transformations is order-dependent. Therefore, the order of application is the same as the order in which they are specified.

Finally, a *VLSI design* is a set of cell instantiations and mask rectangles for use with a specific manufacturing process. Because a cell instantiation references a cell definition and cell definitions may contain instantiations, the structure of a design is hierarchical, having the form of a directed acyclic graph, where the interior nodes correspond to cell definitions and the fringe nodes to mask rectangles. The root set consists of the initial set of cell instantiations and mask rectangles. Thus, a value of the VLSI design vspace is a graph whose structure reflects the designer's viewpoint of the chip to be manufactured. To obtain the manufacturer's viewpoint of the value one must flatten the hierarchy by computing all the instances and sorting the resulting set of rectangles by mask to obtain the individual masks.

Of the two viewpoints, the design viewpoint is the most important because almost all of the effort in creating a chip occurs during the design phase. There are a number of operations that take place during the design of a chip, e.g., mask editing, design-rule checking, compaction, layout and wire routing, circuit extraction and simulation. It is often the case that these operations are performed by different programs, requiring the communication of substantial portions of the design. Because the designer must be able to comprehend the results of any operation and relate them to the original design, the external representation used for communication must reflect the designer's viewpoint, directly or indirectly. Furthermore, VLSI designs are usually quite large and the choice of external representation has a major impact on efficiency. Therefore, the design viewpoint is the basis for the external

```
nmos-vlsi-design = abstract
           rep cif = cif-form[nmos-mask-names]
           rep quad-cif-tree = quad-cif-tree-design[nmos-mask-names]
           rep corner-stitched = nmos-corner-stitched-design
           end

nmos-mask-names = oneof[diffusion:null, polysilicon:null,
                        contact-cut:null, metal:null, implant:null,
                        buried-contact:null ,over-glass:null]
```

Figure 5-8: Vspace definition for VLSI design

representations described in Section 5.2.2 below.

5.2.2 Representations

This section describes three representations for an nMOS VLSI-design vspace. The first is the Caltech Intermediate Form, which is defined in [MC80] and is commonly referred to as CIF. CIF is basically a direct encoding of the abstraction described in Section 5.2.1 above. The second is the Quad-CIF tree which is based on CIF and uses a quad tree to organize the design's layout [Ked82]. The third uses a technique called corner stitching, which is another way of organizing the design's layout [Ous84, Ous85]. This division into three major representations is reflected in the highest level of the abstract vspace definition, which is shown in Figure 5-8.

Note that the CIF and Quad-CIF vspaces in Figure 5-8 are parameterized with respect to a given manufacturing process. Both vspaces represent the design in terms of geometric figures in individual physical mask layers. The actual parameter **nmos-mask-names** is a **oneof** consisting of the physical mask layers that are used to manufacture an nMOS chip. In contrast, the vspace for corner stitching uses abstract masks consisting of physical mask combinations that correspond to the electrical interactions between physical layers. As will be seen later, it is not possible to parameterize the vspace with respect to the manufacturing process.

The major difference between CIF and the other two representations is that the others use an additional data structure to organize the spatial layout. This additional structure is needed to provide fast geometrical operations that are commonly

used by design tools, such as finding all the mask elements in a given region, locating neighbors for stretching, compaction and design-rule checking, and locating empty space for routing. Because Quad-CIF trees and corner stitching provide fast implementations of these operations, they are often used as internal representations by design programs. Providing transfer representations for them allows programs to communicate design data directly. CIF, on the other hand, was initially conceived as a canonical format as shown by the following quotation from [MC80, p. 115].

> "CIF thus serves as the common denominator in the descriptions of various integrated system projects. No matter what the original input methods are . . . , the designs will be translated to CIF as an intermediate, before being translated again to a variety of formats for output devices or other design aids."

The vspace definition for each representation is given in a separate section below. However, before giving the representations I first give definitions for some geometric abstractions that they use.

5.2.2.1 Common Geometric Vspaces

The geometric vspaces that the three representations have in common are defined in Figure 5-9. The most important of these is the `rectangle`, which is an abstract vspace. A rectangle has a position relative to a cartesian coordinate system that is determined by the locations of its four corners. Therefore, an abstract rectangle value consists of four points that form a rectangle. The first two representations, **`centered` and `lower-left`**, specify a rectangle in terms of a reference point, a height and a width. The third representation, `bounds`, specifies maximum and minimum x and y coordinates from which the coordinates for each corner can be computed. As it turns out, these three representations correspond to the preferred representations for the CIF, quad-CIF and corner stitched representations for a VLSI design, respectively.

The length vspace specifies a positive, discrete measure that is specified as a multiple of .01 microns. A point is specified relative to a cartesian coordinate system, and coordinates may be positive or negative. As with length, the unit of measure is

```
rectangle = abstract
            rep centered = struct[height,width:length,
                                  center:point]
            rep lower-left = struct[height,width:length,
                                    lower-left-corner:point]
            rep bounds = struct[x-min,x-max,y-min,y-max:int32]
            end

length = card32
point = struct[x,y:int32]
```

Figure 5-9: Definition of common vspaces

.01 microns. This level of precision is enough to specify the size of mask geometry accurately (at least for the near future).

5.2.2.2 Caltech Intermediate Form (CIF)

The CIF vspace described here and defined in Figure 5-10 is an abstract constructor with two representations, textual and binary. The textual representation is the CIF format as defined in [MC80, pp. 115-126], which is specified by a grammar. It was intended originally to be used with a traditional file system where the files could be concatenated, transferred and read (potentially) by a person. It also has the advantage that designs can be transmitted via electronic mail. The representation given here is restricted to those commands that generate Manhattan geometry. The reader is referred to [MC80] for the details.

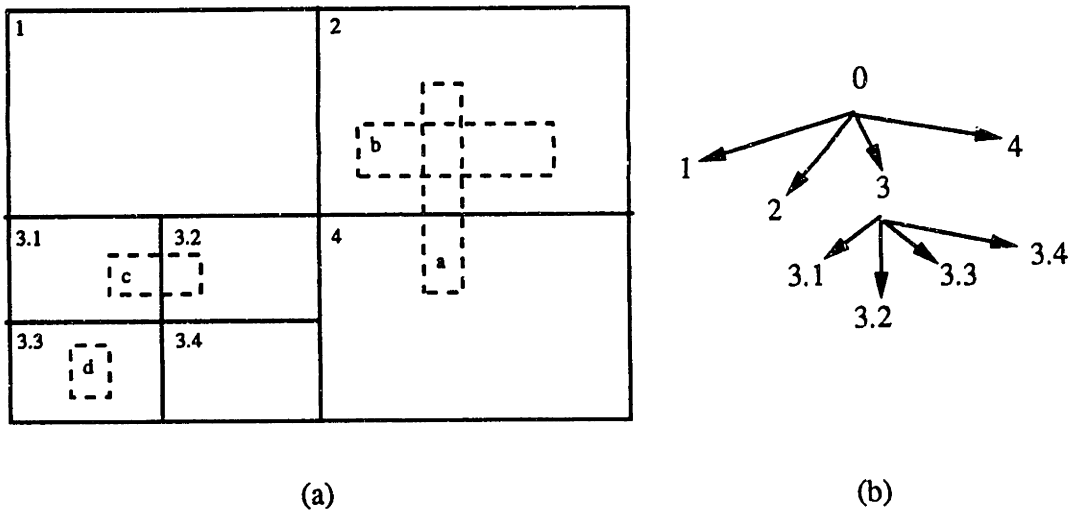
The binary representation is much more in the spirit of distributed systems using RPC for communication. Its definition follows closely the description of the abstraction given in Section 5.2.1. At the top level the value is a sequence of `cif-element`, which corresponds to the root set of the design's graph. The only difference between a `cif-element` here and an element of the abstract value is that a `cif-element` can be a set of rectangles for a single mask as opposed to just one rectangle. This allows some compaction in the size of the representation. The remaining definitions are straightforward translations of the abstract description into a concrete form.

```
cif-form[mask-names:vspace] = abstract
        rep text = sequence32[char]
        rep binary = sequence32[cif-element[mask-names]]
        end

cif-cell-instance =
     struct[parameters:sequence32[cif-parameter],
            reference:container[cif-cell-definition]]

cif-cell-definition = sequence32[cif-element]

cif-element[mask-names:vspace] =
              oneof[cell:cif-cell-instance,
                    geometry:struct[mask:mask-names,
                                    items:sequence32[rectangle]]]

cif-parameter = oneof[translate:point,
                      mirror-x:boolean,
                      mirror-y:boolean,
                      rotation:cif-rotation]

cif-rotation = oneof[r90,r180,r270:null]
```

Figure 5-10: Vspace definition for CIF format

5.2.2.3 Quad-CIF Tree

A quad tree is a data structure that organizes a two-dimensional collection of objects. It is to two dimensions what a binary tree is to one dimension. The root of a quad tree is a single rectangle which is divided into four equal subrectangles. The four subrectangles are the children of the original rectangle. In turn, each child is again subdivided. The process is repeated until there is no cell that is completely contained within a subrectangle (i.e., a rectangle is not subdivided if in the subdivision there are no cells that are completely contained in a child). This is illustrated in Figure 5-11(a), where the solid lines represent rectangles formed by division and the dotted lines represent objects, which are also rectangular. The root rectangle is numbered zero and its children one through four. Further subdivisions are indicated by decimal points. Only the smallest subdivisions have their number given in the upper left



(a) (b)

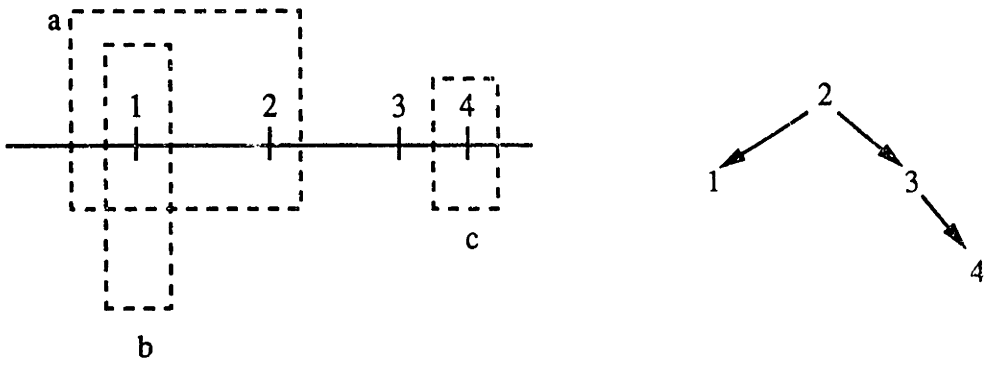
Figure 5-11: Example quad tree structure

hand corner of the rectangle. The corresponding quad tree structure is shown in Figure 5-11(b) where each numbered node corresponds to a rectangle in Figure 5-11(a).

Each object is placed in a node by the following rule. An object is placed in a node if and only if it is completely contained by that node, but is not completely contained by any of its children. For example, in Figure 5-11 object "a" is in node 0, "b" in node 2, "c" in node 3 and "d" in node 3.3. Using this rule, if an object is contained in a node then it must intersect one or both division lines for that node. This property is used to organize the structure of each node in the quad tree.

Within a node there are two binary trees, one for each of the horizontal and vertical division lines. The binary tree represents the successive bisection of the line segment with each node of the binary tree corresponding to a bisection point. An object that intersects the line is placed in the node of the binary tree that corresponds to the highest bisection point in the binary tree that is inside the object. This is illustrated in Figure 5-12, which shows three rectangles intersecting a division line within a node. The numbered tick marks on the line indicate bisection points. The line is only subdivided as far as needed, so the corresponding tree is not necessarily balanced. In this example, rectangle "a" is placed in node 2, "b" in node 1, and "c" in node 4.

If an object in a quad-tree node intersects both division lines, then it is placed



(a) division line with intersecting rectangles (b) corresponding binary tree

Figure 5-12: Example of a binary tree for a division line

in one of the binary trees by convention, e.g., the vertical division line.

In the quad-CIF VLSI representation, every cell definition is represented by a quad tree whose objects are the cell's elements. The quad tree's root rectangle is the smallest rectangle that contains all of the cell's elements. If an element is a mask rectangle then it is represented in the tree by itself. If an element is a cell instance then it is enclosed in a bounding rectangle having the same height and width as the root rectangle of the referenced cell. The location of the bounding rectangle is defined with respect to the coordinate system used by the cell definition. Thus, the bounding rectangle is used to represent the cell instance in the quad tree of the cell being defined.

The vspace definition is given in Figure 5-13 and closely follows the description above. Comparing it with the CIF definition in Figure 5-10, one finds that the main difference is that the vspace for the cell instance in the quad-CIF representation has an additional field **bounds** and that the vspace for the cell definition referenced by the field **reference** is a quad-tree instead of a sequence of elements. The only minor difference of significance is that the top-level element of the quad-CIF design is a single cell instance, unlike the case in CIF where it is a set of elements. The reason is that the top-level elements of a quad-CIF design must be spatially ordered in a quad tree. Because of this difference, conversion from CIF to quad-CIF requires the construction of a dummy cell to hold the top-level elements.

The `quad-cif-tree-design` vspace is a constructor like the `cif-form` vspace

```
quad-cif-tree-design[mask-names:vspace] =
        quad-cif-cell-instance[mask-names]

quad-cif-cell-instance[mask-names:vspace] =
        struct[parameters:sequence32[cif-parameter],
               reference:container[quad-cif-node[mask-names]],
               bounds:rectangle]

quad-cif-node[mask-names:vspace] =
       struct[children:struct[nw,ne,sw,se:quad-cif-edge[mask-names]],
              x-intersect:bin-tree[quad-cif-element[mask-names]],
              y-intersect:bin-tree[quad-cif-element[mask-names]]
              ]

quad-cif-edge[mask-names:vspace] =
        oneof[leaf:null, node:quad-cif-node[mask-names]]

quad-cif-element[mask-names:vspace] =
        oneof[cell:quad-cif-cell-instance[mask-names],
              geometry:mask-rectangle[mask-names]
              ]

mask-rectangle[mask-names:vspace] =
          struct[bounds:rectangle, mask:mask-names]
```

Figure 5-13: Vspace definition for quad CIF tree

encountered earlier. It would also be possible to define a general quad-tree abstract constructor and instantiate it with the quad-cif-element vspace to obtain a quad-CIF-tree. This has not been done here in the interest of keeping the definition short.

5.2.2.4 Corner-Stitching

Corner stitching is another data structuring technique for spatially organizing collections of rectangles. The idea is to divide up an area into a plane of non-overlapping tiles that point to their neighbors. Unlike a quad tree, a corner-stitched plane represents space explicitly in addition to representing objects. The following, more precise, description follows closely that given in [Ous84].

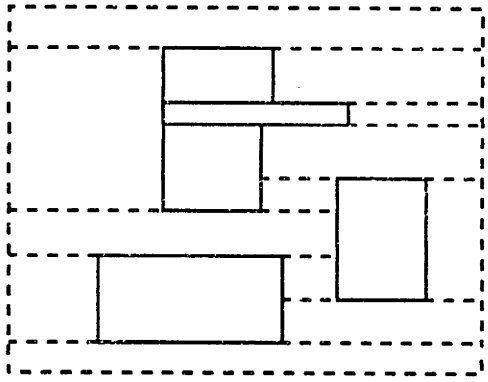


Figure 5-14: Example of a corner-stitched plane

Figure 5-14 shows four objects represented in the corner stitching scheme. The picture resembles a mosaic with rectangular tiles of two types, solid and space. In the figure, the solid tiles are represented with solid lines and space tiles with dashed lines. The tiles must be rectangles with sides parallel to the axes. Tiles contain their lower and left edges, but not their upper or right edges, so every point in the plane is present in exactly one tile. The entire plane is covered from -infinity to +infinity in both x and y (in practice, the largest representable positive and negative numbers are used for the infinities). Coverage to infinity is achieved by extending the outermost tiles, which are always space tiles.

The tiles are organized as maximal horizontal strips, which means that no tile has other tiles of the same type immediately to its right or left. When modifying the data structure, horizontally adjacent tiles of the same type must be split into shorter tiles and then joined into maximal strips. After making sure that tiles are as wide as possible, vertically adjacent tiles are merged together if they have the same horizontal span.

Tiles are linked by a set of pointers at their corners, called *corner stitches.* Each tile contains four stitches, two at its lower-left corner and two at its upper-right corner, as illustrated in Figure 5-15. Tiles that are on the boundary of the area have null pointers to indicate that there is no neighbor (i.e. the tile extends to ±infinity). Since there is one pointer in each of the four directions, the stitches provide a form of sorting.

The vspace representation of a corner-stitched VLSI design is shown in Figure 5-

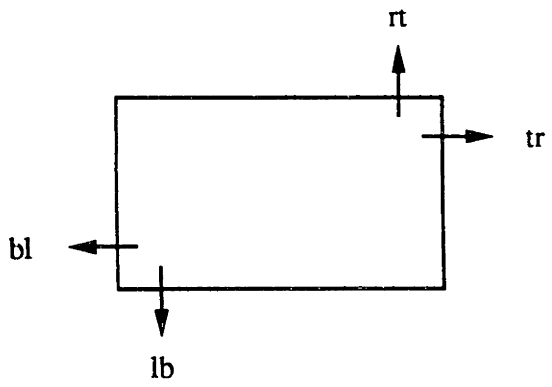


Figure 5-15: Pointers in a corner-stitched tile

16. A cell definition is represented by the vspace `nmos-cs-cell-definition`, which consists of a set of labeled, corner-stitched planes. The plane labeled `subcell` holds the tiles that correspond to cell instances contained by the definition. The remaining planes are determined by the manufacturing process. For an nMOS process there are two mask planes, which are labeled `silicon` and `metal`.

Each plane consists of a set of tiles, where a tile is represented by the vspace `cs-tile`. The parameter `k` in `cs-tile` is used to specify the kind of each tile and is discussed in more detail below. The rectangle defined by the tile is given by the field `location` which is a point corresponding to the lower-left corner of the tile. Because tiles are linked together by container references, an entire plane can be represented indirectly by referencing only one tile.

The first plane is the `silicon` plane, which contains all the tiles that correspond to mask rectangles in all the silicon layers of the device. They are placed in one plane because they interact electrically and must be handled together. The tiles in the silicon plane do not correspond to physical masks, but to abstract masks that represent combinations of physical masks. Abstract masks are necessary because overlapping rectangles from different physical masks must be represented by non-overlapping tiles in the plane.

The concept of abstract masks is illustrated in Figure 5-17. Part (a) shows a rectangle from the polysilicon layer that overlaps one from the diffusion layer. The overlap area forms an enhancement-mode field-effect transistor (efet). The corresponding representation using non-overlapping tiles is shown in part (b) and requires

```
nmos-corner-stitched-design = nmos-cs-cell-definition

nmos-cs-cell-definition =
       struct[silicon:cs-tile[nmos-cs-silicon-tile],
              metal:cs-tile[nmos-cs-metal-tile],
              subcell:cs-tile[nmos-cs-subcell-tile]]

cs-tile[k:vspace] =
       struct[kind:k, location:point,
              stitches:struct[rt:cs-stitch[k], tr:cs-stitch[k],
                              lb:cs-stitch[k], bl:cs-stitch[k]] ]

cs-stitch[k:vspace] = oneof[tile:container[cs-tile[k]], end:null]

nmos-cs-subcell-tile =
       oneof[space:null, cell:container[nmos-cs-cell-instance],
             overlap:nmos-cs-subcell-combination]

nmos-cs-subcell-combination =
       container[unordered-set[container[nmos-cs-cell-instance]]]

unordered-set[v:vspace] = abstract
       rep simple-list = sequence32[v]
       end

nmos-cs-cell-instance =
       struct[transform:cs-transform,
              bounds:rectangle,
              reference:container[nmos-cs-cell-definition]]

nmos-cs-silicon-tile = oneof[space:null, polysilicon:null,
                             diffusion:null, efet:null, ...]
nmos-cs-metal-tile = oneof[space:null, metal:null, contact:null]
```

Figure 5-16: Vspace definition for corner stitching

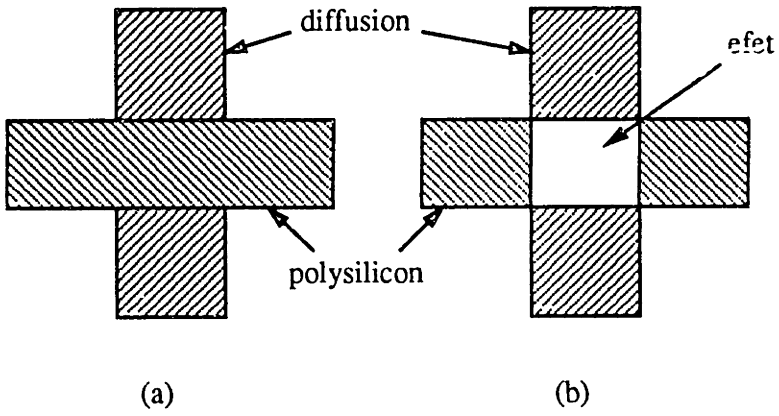


Figure 5-17: Example of an abstract mask

five tiles. The efet tile corresponds to the combination of the polysilicon and diffusion masks. The kind of a silicon tile is given by the vspace `nmos-cs-silicon-tile`, which is only partially written out. In general, each tag of the oneof is a single mask layer or a combination of the mask layers found in the vspace `nmos-mask-names` in Figure 5-8. The tag `space` is used for space tiles.

The second plane is the `metal` plane. It is kept separate from the silicon plane because metal does not interact electrically with silicon except at contact points.

The third plane is the `subcell` plane. Each subcell is specified by a value of the vspace `nmos-cs-cell-instance`, which consists of a reference to a cell definition, a bounding rectangle, and a transform for translating the coordinates of the referenced cell to the coordinate system of the containing cell. The actual definition of the vspace `cs-transform` is not important and so is not given. What is important is that it is not necessarily the same as the transformation specified by the sequence of parameters used by the CIF and quad-CIF representations.

The bounding rectangles are treated like mask rectangles with respect to organizing the subcell plane. That is, an area where two or more bounding rectangles overlap is treated as a new kind of tile. Unlike the other planes, the kind of a subcell tile cannot be specified statically by a fixed set of names. This is reflected in the vspace `nmos-cs-subcell-tile`, which defines a tile as being either a space, a single subcell (or portion thereof), or an area where a set of subcells overlap. The occurances of `container` in the definition mean that there is one kind of tile for

each distinct subcell or distinct combination of overlapping subcells. In the case of overlapping subcells, the vspace `unordered-set` indicates that the tile's kind depends only on the subcells in the set. For this example, the vspace `unordered-set` is an abstract constructor having only one representation.

As a final comment, note that it is not possible to parameterize the definition with respect to the manufacturing process as is the case for the CIF and quad-CIF vspaces. That is because the number of fields in the vspace `nmos-cs-cell-instance` depends on the process. For example, if there were two metal layers instead of one, then there would be an additional field.

5.2.3 Preference List

The important factors that determine the preference list for the VLSI design vspace are the internal representation used by the program and the costs for marshalling between the internal representation and each of the transfer representations. For this example I assume that the internal representation uses a corner stitching data structure and that the machine on which the program runs uses a big-endian representation for scalars such as int32 and card32. The number of transfer representations corresponds to the different possible combinations of representations for the following abstract vspace spaces: `nmos-vlsi-design`, `cif-form`, `rectangle`, and the scalars `card32` and `int32`.

To compare the costs of marshalling for the different external representations, we first look to see if there are vspaces for which the cost difference between representations dominates the total cost. As will be shown below, the choice of representation for the vspace `nmos-vlsi-design` dominates the total cost. Therefore, the discussion first examines the choices for `nmos-vlsi-design` and then considers the choices for `rectangle`, `card32` and `int32`. The choice for `cif-form` is examined as part of the choice for `nmos-vlsi-design`.

5.2.3.1 Comparing costs for `nmos-vlsi-design`

The main difference between the representations for `nmos-vlsi-design` is in the way they represent cell definitions. Because the same number of cell definitions is

representation	encoding	decoding
corner-stitching	$O(T+M)$	$O(T+M)$
CIF, binary	$O(T+C+rR)$	$O(R^{1.5}+S^{1.5})$
CIF, text	$O(T+C+uR)$	$O(R^{1.5}+S^{1.5}+pR)$
quad-CIF	$O(T+C+derR)$	$O(R+S+d^4e)$

T = number of tiles
M = number of entries in subcell map
C = total cost of converting abstract to physical masks
R = number of physical mask rectangles in equivalent CIF form
S = number of subcells instantiated in definition
r = cost of marshalling rectangle in binary form
u = cost of unparsing rectangle
p = cost of parsing rectangle
d = depth of equivalent quad tree ($\approx$5-10)
e = cost to traverse edge of quad tree

Table 5.1: Summary of cost per cell-definition for encoding and decoding

marshaled for a given value in each representation, it suffices to analyze the cost of marshalling a cell definition. Table 5.1 summarizes separately the cost for encoding and decoding a cell definition for each representation. The choice of binary or text representation for `cif-form` is included. The bulk of the following discussion elucidates the information in the Table and discusses the constant factors that are hidden by the "big-O" notation. A summary at the end gives the preferences for encoding and decoding based on the analysis of the constant factors.

To encode a cell definition using corner stitching the program must first traverse each of the silicon, metal and subcell planes, encoding every tile it encounters, and then traverse and encode the cell map. The same is true on decode except that it constructs the planes rather than traverses them.

Encoding a cell definition to either CIF form is complex because abstract tiles must be converted to physical mask rectangles. Referring to Figure 5-17, we see that not only must an abstract tile be decomposed into several physical mask rectangles, but also that adjacent rectangles in the result must be merged. Merging adjacent rectangles significantly reduces the size of the result, which is important given the typical size of a VLSI design. In addition, an unmerged mask results in a fragmented structure, making it more difficult for a designer to understand. The conversion from abstract masks to physical masks and merging the results accounts for the major

encoding costs.

As it turns out, corner stitching is the perfect data structure for merging adjacent rectangles. The program simply creates one corner-stitched plane for each physical mask and, while traversing the original planes, inserts the corresponding physical rectangles for each abstract mask tile into their respective physical planes. The insertion procedure automatically merges adjacent rectangles. The locality that results from inserting the physical rectangles in the order of traversal ensures that the insertion takes linear time. The program then traverses each newly constructed plane and encodes each mask rectangle it encounters. Although it encodes fewer rectangles for transmission than it would abstract mask tiles, the cost of constructing the rectangles is significantly higher than any savings. Therefore, corner stitching is preferred to CIF.

For decoding a cell definition from CIF form, the major cost is constructing the corner-stitched silicon, metal and subcell planes. Because the mask rectangles and subcell instantiations are received in arbitrary order, the expected time to locate the position of each in its corresponding plane is $O(T^{1/2})$ where T is the number of tiles in the plane [Ous84]. Once the position is known the expected time to insert a tile is constant, even if it overlaps other tiles. Because the number of space tiles is at most three times the number of material tiles the expected time to insert n mask rectangles or subcells is $O(n^{1.5})$. Putting this together gives the total shown in Table 5.1. Once again, corner stitching is preferred to CIF.

In the case of textual CIF there is the additional cost of unparsing and parsing the data. The dominant cost of parsing (unparsing) is due to marshalling coordinates and/or lengths in rectangles and is shown by the factor u (p) in Table 5.1.

The above analysis for encoding CIF can also be applied to quad-CIF because the encoding process must first convert to CIF before constructing the quad tree. Therefore, quad-CIF incurs the additional cost of inserting each physical mask rectangle into the quad tree. In the worst case this additional cost is proportional to the tree depth d [Sam88] which is given by the function $\log_4(A_c/A_r)$ where A_C is the area of the cell and A_r is the area of the smallest rectangle. For current VLSI technology d ranges between about 5 and 10. The average case should be better.

In contrast, the analysis for decoding quad-CIF differs somewhat from that for

CIF. It is still the case that the rectangles must be inserted into their respective planes. However, we can take advantage of the spatial locality of the quad tree to reduce the cost of finding the position in which to insert the rectangle. By using the previous rectangle's position as a hint, the expected time to locate the next rectangle's position can be reduced from $T^{1/2}$ to a constant factor[Ous84]. As a result, the total cost is proportional to the number of rectangles, subcells and quad-tree edges traversed. Note that the number of edges in a quad-tree is $O(d^4)$. It should be pointed out that the constants are significant because inserting a rectangle into a plane requires a substantial amount of work; much more so than for decoding the planes directly from a corner-stitched representation.

In summary, by looking at Table 5.1 it is clear that for reasonably sized designs, the order of preference for decoding a VLSI design is a total order with corner stitching first, quad-CIF second, CIF-binary third and CIF-text last. With respect to encoding it is clear that corner stitching is preferred to CIF, CIF-binary to CIF-text and CIF-binary to quad-CIF. It is not clear how CIF-textual compares to quad-CIF because the factor $derR$ for quad-CIF is worst case whereas the factor uR for CIF-textual is an actual cost. Because of this we cannot state a preference between the two. In the case where one preference list must be used for both encoding and decoding, then the decoding preferences should be used because the non-linear factors become significant.

5.2.3.2 Comparing costs for `rectangle` and scalars

The first point to notice here is that not all combinations of representations for `card32` and `int32` make sense. A system that uses big-endian (little-endian) for one will use big-endian (little-endian) for the other. Because marshalling a VLSI design is mostly marshalling rectangles, it is important to optimize the code for these cases. If it is necessary to include cross combinations of big and little-endian (for the rare case), then they can be left unoptimized and included at the end of the preference list.

The second point is that the `bounds` representation for `rectangle` does not use `card32`. Because contexts may not contain irrelevant assignments, any context that assigns `bounds` to `rectangle` must not have `card32` in its signature unless it is

needed for some other vspace in the interface.

With these two points in mind it is easy to perform the analysis. With respect to `rectangle`, the most preferred representation is `bounds` because the internal representation only stores the coordinate of the lower-left corner. Computing the bounds requires following only two stitches to look up the maximum x and y values. Converting to `lower-left` requires two additional subtractions to compute the height and width. Converting to `centered` requires computing the height and width as well as relocating the reference point which, in this case, is achieved by shifting right the height and width one bit (to divide by two) and adding the result to the x and y components of the coordinate. So the order of preference for `rectangle` is `bounds`, `lower-left` and `centered`, and the difference between any two is only about 2 to 8 CPU cycles.

Each rectangle has four scalars so the effect of the difference between big-endian and little-endian is multiplied by four. Given that it takes about 6 to 10 CPU cycles to swap bytes, the cost difference for scalars dominates that for rectangles. The preference list for scalars then is:

```
let
  rect = rectangle
  be = big-endian
  le = little-endian
in
  {rect:bounds,int32:be}
  >(({rect:lower-left} > {rect:centered}) || {card32:be,int32:be})
  >{rect:bounds,int32:le}
  >(({rect:lower-left} > {rect:centered}) || {card32:le,int32:le})
```

As a final remark, note that the difference in cost between any two assignments in the preference list above is on the order of about 20 to 40 CPU cycles per rectangle. This difference is dominated by the differences that result from different choices for `nmos-vlsi-design` discussed earlier. The fact that `nmos-vlsi-design` dominates makes it seem that quibbling over a few instructions per rectangle is irrelevant. However, a VLSI design consists mostly of rectangles, usually thousands of them, and the total difference can be significant.

5.2.3.3 Putting it all together

The previous sections have given quite a bit of analysis. Putting it all together and assuming that VLSI designs are both sent and received gives the preference list shown in Figure 5-18. The overall picture is summarized in the following points along with a few observations.

1. The choice of representation for `nmos-vlsi-design` dominates the overall cost. For encoding, corner stitching is preferred to CIF-binary, which in turn is preferred to CIF-text and quad-CIF. CIF-text and quad-CIF are unordered. For decoding, the order of preferences is corner stitching, quad-CIF, CIF-binary and CIF-text.

2. The representation for corner stitching uses `rectangle`, `point` and `card32`. The preferences for `point` are already included in the preference list for `rectangle`. Preferences for `card32` are added by joining them to those for rectangle.

3. CIF-text uses neither scalars nor rectangles, but does use `char`. I assume that the program supports ASCII and EBCDIC representations with ASCII being preferred.

4. Cross combinations of representations for `int32` and `card32` are not considered.

```
let
  vlsi = nmos-vlsi-design
  corner-stitch = nmos-corner-stitched-design
  cif = cif-form
  quad-cif = quad-cif-tree-design
  be = big-endian
  le = little-endian
  rect = rectangle
  rect-pref =
      {rect:bounds,int32:be}
      > (({rect:lower-left} > {rect:centered}) || {card32:be,int32:be})
      > {rect:bounds,int32:le}
      > (({rect:lower-left} > {rect:centered}) || {card32:le,int32:le})
in
  ({vlsi:corner-stitch} || rect-pref || ({card32:be,int32:be}
                                        > {card32:le,int32:le}))
  > (({vlsi:quad-cif} > {vlsi:cif,cif:binary}) ≫ rect-pref)
  > ({vlsi:cif,cif:text} || ({char:ascii} ≫ {char:ebcdic}))
```

Figure 5-18: Preference list for VLSI design

Chapter 6

Implementing Context Lists

The primary computation during negotiation is determining the best context given two context lists, where one has priority (see Section 3.3). The computation is made by the presentation layer of a program when it receives a context list from a peer. A schematic of the basic system structure is shown in Figure 6-1, where the arrows indicate the direction of information flow.

At compile time, the *presentation compiler* takes a *context list expression* provided by a programmer and constructs an *internal representation* of the context list. The compiler communicates with the registry (as indicated by the zig-zag arrow) to look up identifiers for vspaces, vspace components and representations. The internal representation is then linked with the code of the program represented in the figure by the box labeled *presentation layer.*

At runtime, a context list is received from a peer, encoded in some transfer representation. The context list must be decoded into the internal representation used by the presentation layer before the best context can be computed.

In this chapter I examine algorithms for computing the best context and compiling context lists. In particular, I examine two algorithms for computing the best context. Although both algorithms use a graph for the internal representation, the form of the graph is very different in each case. Each algorithm has its advantages and disadvantages. In both cases, the transfer representation is a direct encoding of the internal representation. This minimizes the work required to decode a context list on reception.

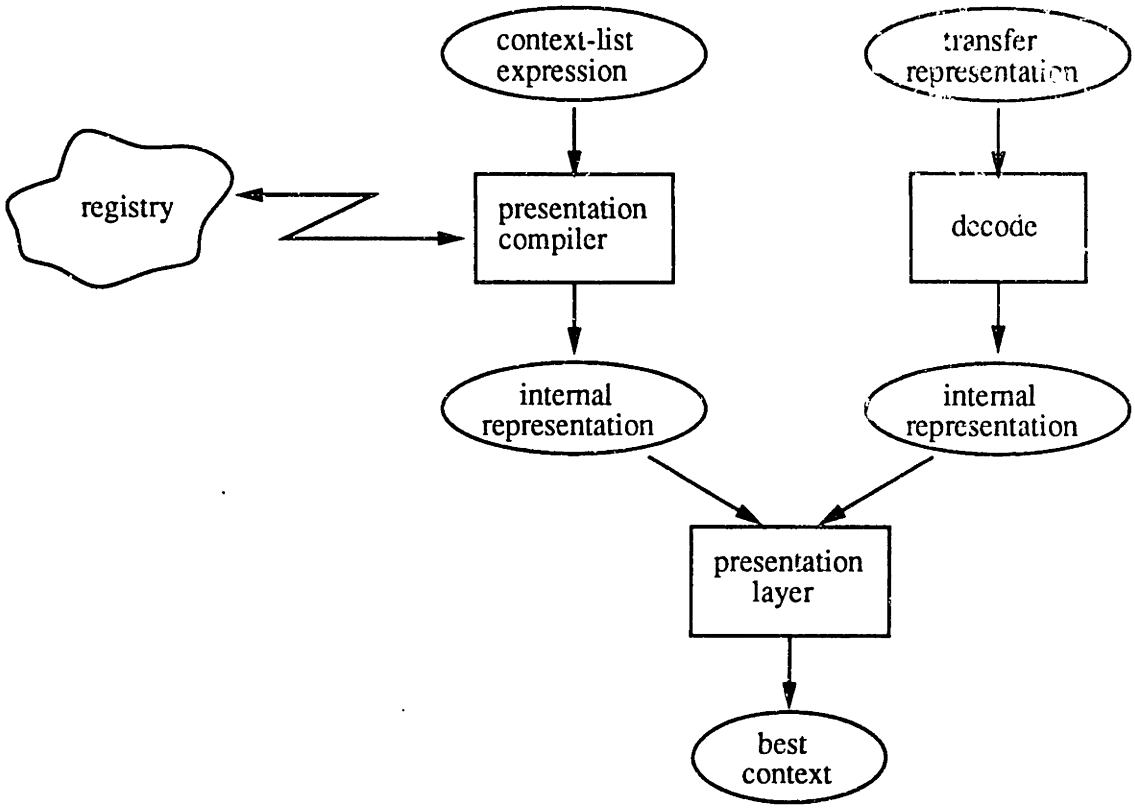


Figure 6-1: Overview of the computation for the best context

I also present algorithms for compiling a context list expression into an internal representation. These algorithms are embedded in the operations of two key data abstractions: the preference list abstraction and the assignment abstraction.

The algorithms described in this chapter have been implemented in prototype form and their performance measured using several example context lists. The main purpose of these measurements is to show that negotiation with context lists is feasible.

The remainder of this chapter is organized as follows. Section 6.1 introduces some of the notation used to describe the algorithms. Section 6.2 describes in detail the two algorithms for computing the best context. Section 6.3 describes how contexts are represented and introduces the concept of a context UID. Sections 6.4 and 6.5 give a detailed description of the preference list and assignment data abstractions. Section 6.6 gives an overview of the prototype implementation. Section 6.7 gives execution times for computing the best context given two context lists. It also

examines the tradeoffs between the two algorithms for computing the best context.

6.1 Some Notation

The algorithms in the following sections are written using a "semi-functional" language with high-level data types such as sets, sequences and functions. The language is functional in that operations do not mutate their arguments; instead, they create new values. However, the language is not completely functional because it allows assignment (which is denoted by $:=$). Therefore, the order of the statements in an algorithm is important. The language is simple enough that only the notation for a few operations needs to be explained.

The most common data type is $SET[T]$, where T gives the type of the set's elements. Sets have the usual operations of $\in$, $\cup$, $-$, and $\cap$, along with the operation $choose(S)$, which chooses some element of S.

For functions, there is function application, e.g., $f(x)$, and function extension, which is expressed as $f[x \mapsto v]$, read as "extend f at the point x to have value v," and defined by the following:

$$f[x \mapsto v](y) = \begin{cases} v & \text{if } y = x \\ f(y) & \text{otherwise} \end{cases}$$

The x in the above formula may be replaced with the symbol "$*$," which means extend the function at every point, e.g., $f[* \mapsto 2]$ denotes the constant function whose value is 2 everywhere. In addition, the v may be omitted, which means that x is removed from the domain, i.e., $f[x \mapsto](y)$ is undefined if $y = x$, and $f[x \mapsto](y) = f(y)$ if $y \neq x$. Combining the two gives $f[* \mapsto]$, which denotes the function whose domain is empty, i.e., if $g = f[* \mapsto]$, then $g(x)$ is undefined for all x.

Finally, sequences are treated the same as in Amalgam, e.g., $s[i]$ is the i^{th} element of sequence s, where $1 \leq i \leq length(s)$.

6.2 Computing the Best Context

As explained in Section 3.3, finding the best context given two context lists involves computing the function $max_contexts(C, S)$, which gives the maximum contexts in

context list C that are in also in the set S. If C_1 and C_2 are context lists with C_1 having priority then the best contexts are given by equation 3.6, which is repeated here:

$$best_contexts = max_contexts(C_2, max_contexts(C_1, dom(C_2)))$$

Because context lists are partial orders there may be more than one best context, in which case one is chosen arbitrarily.

A context list is represented abstractly by a directed, acyclic graph (DAG) of the form $\langle V, E \rangle$, where V is the set of vertices and E the set of edges. If $G = \langle V, E \rangle$ is a DAG, then G *represents context list* C if and only if the following is true:

$$(dom(C) = V) \wedge \forall x, y \in V \,.\, \langle x, y \rangle \in \mathcal{R}_C \Leftrightarrow \text{there is a path from } x \text{ to } y \text{ in } G \quad (6.1)$$

In this section I examine two different representations for a DAG: an adjacency list and an equivalence-class list. Each representation has its own algorithm for computing *max_contexts*. These two representations provide a tradeoff between space and time. An adjacency list gives a more compact representation for large context lists, while equivalence-class lists are generally more efficient when computing the best context.

Both algorithms optimize two common cases. The first is where both applications prefer the same context. The second is where one application supports the preferred context of the other. These cases are common because a given application program is typically optimized for use with other programs.

The remainder of this section is divided into two parts. The first describes the adjacency-list representation and the second the equivalence-class representation. Each section defines a representation, describes an algorithm for *max_contexts*, and addresses implementation issues that make the algorithm efficient.

6.2.1 Adjacency-List Representation

An adjacency list representation of a DAG is given by the abstract data type *Graph*, which is a function from *Context* to *Set*[*Context*]. If G is a function of type *Graph*, then the set of vertices V of the corresponding DAG is equal to the domain of G, and the set of edges E is equal to the set $\{\langle x, y \rangle\}$, where x is in the domain of G and $y \in G(x)$. The following operations are defined on a value G of type *Graph*:

```
max_contexts = proc(G:Graph, S:Set[Context]) returns(Set[Context])
1   R:Set[Context] := ∅
2   pred_exam:Context → Int := pred_exam[* ↦ 0]
3   pending:Set[Context] := roots(G)
4   while pending ≠ ∅ do
5         v := choose(pending)
6         pending := pending − {v}
7         if v ∉ S
8           then for w ∈ G(v) do
9                     pred_exam := pred_exam[w ↦ pred_exam(w) + 1]
10                    if pred_exam(w) = indegree(G,w)
11                      then pending := pending ∪ {w}
12          else R := R ∪ {v}
13  return R
```

Figure 6-2: Algorithm for *max_contexts* using an adjacency list

$dom(G)$ = the set of vertices of G.

$G(x)$ = the successors of vertex x.

$roots(G)$ = the set of vertices $x \in dom(G)$ that are roots of G.

$indegree(G,x)$ = the in-degree of vertex x

The algorithm computing *max_contexts* is given in Figure 6-2. It takes a context list represented by G and a set of contexts S as arguments, and returns a set consisting of the contexts in S that are maximal in G. The basic idea of the algorithm is to enumerate the contexts (i.e., vertices) of G in topological order and check them for membership in S. Topological order corresponds to the order of preference: if context x is preferred to y then x is enumerated before y. The reason for enumerating contexts in topological order is to avoid examining contexts not potentially in the result. For example, if x is preferred to y and $x \in S$, then there is no need to look at y. To avoid looking at y, the algorithm "prunes" the subgraph rooted at x. The trick is to interleave enumeration with pruning.

The algorithm enumerates the contexts in topological order by keeping track of how many predecessors have been examined for each context. This is the purpose of the function *pred_exam*, which is initialized to all zeroes in line 2. The set *pending* holds all the contexts that must be examined for membership in S. It is initialized to the roots of G in line 3. The while loop in lines 4–12 enumerates the contexts and

examines them. It first chooses a context v from *pending* that must be examined and then deletes it. If the context is *not* in S, then it increments the number of predecessors examined for each successor of v. Any successor w whose predecessors have all been examined is inserted into *pending* in lines 10–11. If the context is in S, then it is simply inserted into the result R in line 12.

The algorithm examines contexts in topological order because a context is inserted into *pending* only after all of its predecessors have been examined. In addition, it never examines any context w in the subgraph rooted at any context $v \in R$ because none of v's successors is inserted into *pending*. Finally, all contexts in S that are maximal in G are enumerated and inserted into the result R because all their predecessors are visited and discarded.

6.2.1.1 Implementation Issues

A number of implementation details must be addressed in order to make the algorithm efficient. The following lists some of the more important ones:

1. Eliminate transitive edges. Although the algorithm performs correctly in the presence of transitive edges, they are redundant and make the computation take longer. Transitive edges can be removed by the context-list compiler.

2. Make $G(x)$ fast by providing "direct links" from each vertex to its successors.

3. Make $indegree(G, x)$ fast by computing it at compile time and storing it along with the vertex that represents x. The in-degree of each vertex should be part of the transfer representation so that the receiver of a context list need not compute it as part of negotiation.

4. Make $roots(G)$ fast by computing it at compile time and storing it along with the graph. The root set should also be an explicit part of the transfer representation.

5. Make $pred_exam(x)$ fast by associating an index with each vertex and using an array. The array can be reused for successive invocations.

6. Implement the set *pending* as a queue. Because the order in which vertices are chosen is not important, a queue gives the most efficient implementation.

7. Implement S and R as hashed sets. The most common operation performed on S is testing for membership. A hashed set makes this efficient. R should be a hashed set because it can be passed to a subsequent invocation of *max_contexts*.

Points 1, 3, 4, and 5 are performance issues that are best addressed by choosing a good representation for a context list. As mentioned at the beginning of this chapter, the transfer representation is a direct encoding of the internal representation. Therefore, ony a transfer representation is given here. One such representation is given by the following vspace definitions:

context-list = **struct**[num-roots:**card32**, vertices:sequence32[vertex],
successors:sequence32[adjacency-list]]
vertex = **struct**[label:context, indegree:**card32**]
adjacency-list = sequence32[**card32**]

The field *num-roots* gives the number of roots in the graph. The field *vertices* lists all the contexts in the graph. The index of a context in the sequence is the index associated with that context's vertex (point 5). The first *num-roots* contexts in the sequence are the root vertices (point 4). The remaining vertices can be ordered arbitrarily. The field *successors* gives the adjacency list for each corresponding context in *vertices*. The field *indegree* gives the in-degree for each corresponding vertex (point 3). To meet point 1, any value of the above vspace must not have any transitive edges.

The above definition gives an abstract representation for a context-list. To obtain a transfer representation, one must choose (transfer) representations for the components sequence32 and **card32**.

The number of bytes required to store an internal representation of the above vspace is easy to calculate, assuming that **int32** and **card32** require 4 bytes, and sequence32 requires one 4-byte word to store the length in addition to the storage needed for the elements. If we also include the storage needed by the comparison algorithm for the array *pred_exam*, then the size of the corresponding internal representation is given by the following equation:

$$storage(G) = vc + 4(2v + e + 3) \tag{6.2}$$

where c is the number of bytes for a context, v the number of vertices, and e the number of edges. The term vc is the storage required for the field "label" in the vspace "vertex," and there are v occurrences of a vertex. The factor 4 is the number of bytes in a **card32**. The formula inside the parenthesis is in terms of **card32**. One half of the term $2v$ is due to the field "indegree" in the vspace "vertex." The other half of the $2v$ is due to the length part of the vspace "adjacency-list." The term e is the number of elements in all the adjacency lists. The constant 3 comes from the vspace "context-list" and includes the field "num-roots" and the length part of the fields "vertices" and "successors."

One last implementation trick is to avoid initializing every entry in *pred_exam* each time *max_contexts* is invoked. This can be done by using an additional set *touched* to keep track of all the elements of *pred_exam* that are modified. The modified entries are cleared before returning. The corresponding algorithm is shown in Figure 6-3, where the numbered lines correspond to the lines in Figure 6-2 and the unnumbered lines are new. The allocation and initialization of *pred_exam* are not shown. Like *pending*, *touched* can be implemented efficiently as a queue. The modified algorithm optimizes the case where the contexts in the result are "near" the roots. In particular, it optimizes the case where the maximal elements are contained in the set S.

6.2.2 Equivalence-Class List Representation

An *equivalence-class list* is a tuple $\langle T, Q \rangle$, where T is a sequence of $Context$ and Q is a sequence of $Set[Int]$. Informally, T represents a topological sort of a context list C, where $T[i]$ is the i^{th} context in the sort. $Q[i]$ represents the equivalence class consisting of the set of contexts $T[j]$ that are equivalent to the context $T[i]$. $Q[i]$ contains the indices j of the contexts $T[j]$ instead of the contexts themselves. If L is an equivalence-class list, then its topological sort is denoted by $L.T$ and its equivalence classes by $L.Q$.

More formally, if C is a context list, then an equivalence-class list L *represents* C if and only if the following formulas are satisfied:

$$dom(C) = elements(L.T) \tag{6.3}$$

```
max_contexts = proc(G:Graph, S:Set[Context]) returns(Set[Context])
1   R:Set[Context] := ∅
2
3   pending:Set[Context] := roots(G)
    touched := ∅
4   while pending ≠ ∅ do
5         v := choose(pending)
6         pending := pending − {v}
7         if v ∉ S
8           then for w ∈ G(v) do
                      if pred_exam(w) = 0 then touched := touched ∪ {w}
9                     pred_exam := pred_exam[w ↦ pred_exam(w) + 1]
10                    if pred_exam(w) = indegree(G, w)
11                      then pending := pending ∪ {w}
12          else R := R ∪ {v}
    for v ∈ touched do pred_exam := pred_exam[v ↦ 0]
13  return R
```

Figure 6-3: Modified algorithm for *max_contexts* using *touched*

$$\langle L.T[i], L.T[j] \rangle \in \mathcal{R}_C \Rightarrow i < j \tag{6.4}$$

$$\forall (1 \leq i, j \leq size(dom(C))) \,.\, (i \in L.Q[j] \Leftrightarrow (\langle L.T[i], L.T[j] \rangle \notin \mathcal{R}_C \wedge \langle L.T[j], L.T[i] \rangle \notin \mathcal{R}_C)) \tag{6.5}$$

where $size(dom(C))$ gives the number of contexts in C.

Formulas 6.3 and 6.4 say that $L.T$ represents a topological sort of $\mathcal{R}_C$ in that, if x is preferred to y, x comes before y in the sequence $L.T$. Formula 6.5 defines the "equivalence class" of each context in C. This is not an equivalence class in the usual sense where equivalence connotes equality. Here, $i \in L.Q[j]$ if and only if the contexts $L.T[i]$ and $L.T[j]$ are not ordered by $\mathcal{R}_C$. In terms of choosing a context, it means that $L.T[i]$ is "equally preferred" to $L.T[j]$ because neither is preferred to the other. However, the relation "equally preferred" is not transitive. Therefore, if $i \in L.Q[j]$ and $k \in L.Q[j]$, it is not necessarily the case that $i \in L.Q[k]$ or $k \in L.Q[i]$.

The algorithm that computes *max_contexts* is given in Figure 6-4. It takes a context list represented by L and a set of contexts S as arguments, and returns a set consisting of the contexts in S that are maximal in L. The algorithm consists of two phases corresponding to lines 2–5 and 6–12. The first phase scans the topological sort in order until it finds the first context that is in S. Call this first context f.

```
max_contexts = proc(L:ECList, S:Set[Context]) returns(Set[Context])
1   R:Set[Context] := ∅
2   i:Int := 1
3   while L.T[i] ∉ S ∧ i ≤ size(L.T) do  i := i + 1
4   if i > size(L.T) then return ∅
5   R := R ∪ {L.T[i]}
6   E:Set[Int] := L.Q[i]
7   while E ≠ ∅ do
8         j:Int := least(E)
9         E := E − {j}
10        if L.T[j] ∈ S then
11           R := R ∪ {L.T[j]}
12           E := E ∩ L.Q[j]
13  return R
```

Figure 6-4: Algorithm for *max_contexts* using an equivalence-class list

Line 5 inserts f in the result R.

The second phase scans the equivalence class of f in topological order, looking for additional contexts that are in S. The operation *least* in line 8 chooses the next element in the topological sort of the corresponding contexts in E. Call this element y. If y *is not* in S then we continue scanning; if y *is* in S then it is inserted into the result. In addition, any contexts that are less preferred than y must be removed from E. This is the purpose of line 12. An example illustrating the need for this step is shown in Figure 6-5. The contexts in the figure are numbered in topological order (i.e., $T[i] = ci$). In this example, $max_contexts(C, S) = \{c2, c3\}$, $f = c2$ and $L.Q(2) = \{3, 5\}$. Even though $c5$ is in $c2$'s equivalence class, it is not in the result because $c3$ is preferred to $c5$. If $L.Q(2)$ were not scanned in topological order, then $c5$ would be inserted into the result, which is incorrect.

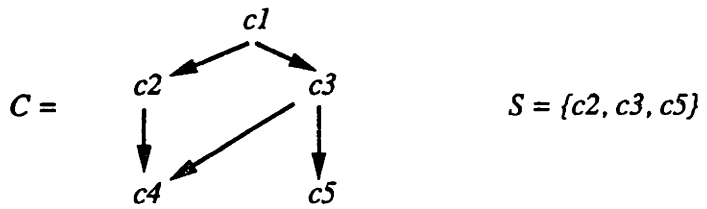


Figure 6-5: Example graph for equivalence-class list algorithm

Topological Sort	Equivalence Matrix
$c1$	0 0 0 0 0
$c2$	0 0 1 0 1
$c3$	0 0 0 0 0
$c4$	0 0 0 0 1
$c5$	0 0 0 0 0

Figure 6-6: Example equivalence-class representation

6.2.2.1 Implementation Issues

In order for the algorithm in Figure 6-4 to be efficient, both the *least* operation in line 8 and the intersection in line 12 must be efficient. The best representation for an equivalence class requires a compromise between these two operations. For *least*, the best representation is an ordered sequence of integers. However, this would make intersection slow. The best compromise is to use a bit vector. The bit vector is scanned in order to find successive one bits, while intersection is accomplished by taking the logical "and" of two vectors.

We can reduce the number of bits scanned in a bit vector by noting that, if $L.T[i]$ is the context found in the first phase, then it is not necessary to examine any $j \in L.Q[i]$ in the second phase if $j \leq i$. The reason is that the corresponding contexts $L.T[j]$ were already examined in the first phase. Therefore, if E is a bit vector representing $L.Q[i]$, then $E[j] = 1$ if and only if $j \in L.Q(i)$ and $j > i$, and $E[j] = 0$ if and only if $j \notin L.Q(i)$ or $j \leq i$.

$L.Q$ is then represented by a sequence of bit vectors whose organization parallels that of the sequence $L.T$ in that the bit vector $L.Q[i]$ represents the equivalence class for context $T[i]$. The sequence of bit vectors forms an n by n matrix called an ***equivalence matrix.*** Figure 6-6 gives the topological sort and equivalence matrix for the example context list in Figure 6-5. Because $L.Q[i][j] = 0$ when $j \leq i$, the matrix is an upper triangular matrix. This property can be used to reduce the amount of storage required for an equivalence matrix.

It is not clear what the best transfer representation for an equivalence-class list is at this point. In the examples studied in Section 6.7, there are often many zeroes at the end of a row. By keeping track of the highest one bit in each row, an

implementation can reduce the number of bits that must be scanned in the second phase of the algorithm. The examples also exhibit runs of zero bits and one bits, which suggests that run-length encoding may reduce the size of a bit vector as well as the time required to scan it. Whether or not these properties hold for context lists in general requires further research with real applications.

The following vspace definitions give a representation that closely matches the internal representation used in the prototype implementation described in Section 6.6:

context-list = **struct**[contexts:sequence32[context],
classes:sequence32[bit-vector]]
bit-vector = **struct**[first:int32, last:int32, vector:sequence32[int32]]

The fields *first* and *last* give the index of the first and last one bit in the vector. The field *vector* represents the actual bits. It is aligned on a 32 bit boundary so that the first bit in the first **int32** of the sequence has the index $32((first-1)/32)+1$, and the last bit in the last **int32** of the sequence has the index $32((last+31)/32)$, where the division by 32 is integer division. A transfer representation for the above vspace would require specifying how an **int32** represents a vector of 32 bits, as well as choosing a (transfer) representation for **int32** and sequence32.

The actual representation used in the prototype implementation differs from that described above in that it does not eliminate the zeros below the diagonal of the equivalence matrix.

The number of bytes required to store an internal representation of the above vspace depends on the individual context list. In any case, the maximum is given by the following equation:

$$storage(L) = nc + 4(2 + 3n + ((n-1) \bmod 32) + 1 + \sum_{i=2}^{((n-1)/32)+1} 32i) \qquad (6.6)$$

where n is the number of contexts in L, c is the number of bytes needed to store a context, and division is integer division. The term nc gives the storage required for the elements in the field "contexts" in the vspace "context-list." The factor 4 is the number of bytes in a **card32** or **int32**. The formula contained in the outermost set of parentheses is in terms of **card32** and **int32**. The term 2 is for the length of the sequences in the fields "contexts" and "classes." There are n bit-vectors, and the

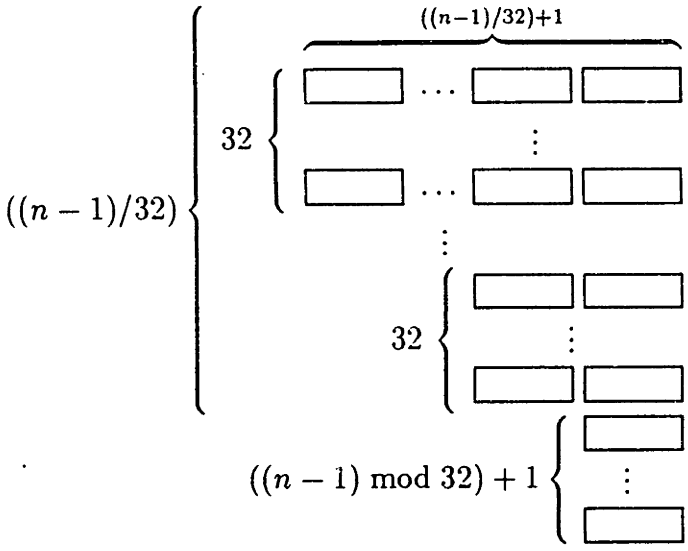


Figure 6-7: Storage for equivalence-class matrix

term $3n$ covers the storage for the fields "first" and "last" in "bit-vector," together with the length part of the sequence in the field "vector."

The remainder of the formula, starting with $((n-1) \bmod 32)$, gives the total number of elements in all the bit-vectors. It corresponds to an upper triangular matrix. This is illustrated in Figure 6-7, where a rectangular box in the figure corresponds to an **int32** that represents 32 bits. The last $((n-1) \bmod 32)+1$ rows contain only one **int32** element each. The summation corresponds to the preceding rows. Each group of 32 rows contains one more **int32** then the group that follows it. There are $(n-1)/32$ such groups.

6.3 Representing Contexts

The algorithms in the previous section make no assumptions about how a context is represented in a context list. They only require that the *insert* and *member* operations on sets of contexts be efficient. This suggests using a hash table to implement a set of contexts.

One possible representation for a context is an assignment. In this case the transfer representation of a context is similar to the representation described for assignments in Section 6.5. Although assignments can be implemented relatively

efficiently, the time to compute a hash function is proportional to the number of components in the assignment. In the worst case, every context of a received context list must be hashed in order to compute the best context. This can introduce substantial overhead.

Another representation for a context is a *context UID*, which is a unique identifier that denotes a context for a specific negotiable set. The registry is responsible for assigning context UIDs to contexts. A context UID is formed by concatenating a small identifier (e.g., a 32-bit integer) to the UID of the negotiable set. The small identifier need only be distinct with respect to other contexts for that negotiable set. Because context lists are specific to a negotiable set, the contexts within a context list can be represented using only the small identifier. This permits a very efficient implementation for a set of contexts.

Context UIDs are clearly preferred over assignments because they are both more compact and faster to hash. However, not all systems are capable of assigning context UIDs to all contexts, as discussed later in Section 7.2. These systems must use assignments to represent contexts.

6.4 The Preference-List Abstraction

We now shift our focus from computing the best context to compiling context-list expressions into the internal representation used by the presentation layer. Recall from Chapter 3 that a context list for a negotiable set is a preference list, where every assignment in the preference list is a context for that negotiable set. Therefore, compiling context lists consists basically of implementing the preference-list operations in Chapter 4. It is important that these operations be implemented efficiently. A naive implementation can result in long compilation times, making compilation a tedious chore for the programmer.

The operations of the preference-list abstract data type consist primarily of the union and join operations from Section 4.3.1, which are used to construct preference lists from expressions. Once a preference list is constructed, the assignments are checked to verify that they are indeed contexts, and the result is converted into a context list for use by the presentation layer during negotiation.

A preference list is represented abstractly by a DAG in the same way as a context list. If $G = \langle V, E \rangle$ is a DAG, then G *represents preference list* P if and only if the following is true:

$$(dom(P) = V) \wedge \forall x, y \in V \,.\, \langle x, y \rangle \in \mathcal{R}_P \Leftrightarrow \text{there is a path from } x \text{ to } y \text{ in } G \quad (6.7)$$

The best representation for a preference list is an adjacency list. A preference list is represented by the abstract data type *Graph* whose definition is similar to that on page 114 of Section 6.2.1, except that here it is a function from *Assignment* to *Set*[*Assignment*] instead of from *Context* to *Set*[*Context*].

Ideally, the representation would be a reduced graph, i.e., it would contain no transitive edges. The reason is that the running times of the join operations are proportional to the number of edges; transitive edges represent redundant information. As shown later, it is impossible to build a reduced graph directly from reduced operand graphs in certain cases. The alternatives are: 1) reduce after every operation that can introduce a transitive edge, or 2) ignore transitive edges. The first alternative is inefficient for small graphs, while the second is inefficient for large graphs. If the second alternative is used, the representation must be reduced when it is converted into a context list that is represented by an adjacency list. The prototype implementation uses the second alternative because the context lists for the examples in Section 6.7 are not large enough to introduce significant overhead.

There are five preference list operations that are used to construct preference lists from expressions: create singleton, conjunction, dominant union, consistent join, and dominant join. The create singleton operation constructs a preference list containing exactly one assignment. Its implementation is trivial and not described here. The other four operations are described in the remainder of this section. In each case the description notes whether or not transitive edges can be introduced, and if so, then how. Each description also gives the worst-case running time using big-O notation. The following variables are used in the formulas:

v_i = number of vertices (i.e., assignments) in operand i's graph.

e_i = number of edges in operand i's graph.

l = maximum length of assignments in an operand.

Any additional variables are defined at the point of use.

6.4.1 Conjunction

The conjunction operation takes two operands and mutates the first operand to produce the result. The implementation consists of a number of relatively simple steps. The first step checks that every assignment in the domain of one operand is either equal or comparable to every assignment in the domain of the other (see Section 4.3 for a definition of comparable). While checking, it computes the union of the vertices and identifies vertices from the operand graphs that are equal (i.e., that have the same label). The second step forms the union of the two graphs by inserting all the vertices and edges from the second operand into the first. An edge is not inserted if it would result in a duplicate edge. In addition, a root vertex in the first operand must be removed from the root set if it is the target of a newly inserted edge. Likewise, a root from the second operand must be inserted into the resultant root set if it is distinct from any vertex in the first. The last step checks the resultant graph for cycles using a depth-first search. The presence of a cycle means that the information in the two operands is inconsistent. An exception is signaled if a cycle is found.

The running time is $O(v_1 v_2 l)$ for checking comparability, and $O(v_1 + v_2 + e_1 + e_2)$ for both the union and the check for cycles. The predominant cost is checking comparability.

Note that the resultant graph can have transitive edges. Because conjunction is used typically at the lowest levels in a preference-list expression and involves only a few assignments, the number of transitive edges introduced is small, if any.

6.4.2 Dominant Union

The dominant-union operation takes two operands and mutates the first operand to produce the result. The implementation of the dominant union is straightforward. It first checks that the domains of the operands are disjoint and that every assignment in the domain of one operand is comparable to every assignment in the other. While checking, it makes a list of all leaf vertices in the first operand. It then forms the union of the two graphs by inserting all the vertices from the second operand into the graph of the first operand. Finally, it inserts an edge from every leaf vertex in

the first operand to every root in the second.

The running time is $O(v_1 v_2 l)$ for checking that the two operands are disjoint and comparable. The actual union requires $O(v_2 + v_l v_r)$ time, where v_l is the number of leaves in the first operand and v_r the number of roots in the second. The initial checks constitute the predominant cost.

6.4.3 Consistent Join

Computing the join directly from the operand's graphs is undoubtedly the most difficult operation. The main problem is how to handle the case when there are assignments in one operand that do not join with any assignment in the other. The approach taken here is to compute the graph as if all assignments did join and then "fix up" the graph to remove those that did not.

Explaining this approach requires quite a bit of elaboration. The following description begins by first assuming that all the assignments join. It gives a mathematical description of the graph to be computed and shows that it is correct. In addition, it shows that the result is a reduced graph if both operands are also reduced graphs. It then describes the algorithm that computes the result. Finally, it shows how to fix up the graph to remove vertices that did not join.

Let X and Y be preference lists that are represented by the graphs $G_X = \langle V_X, E_X \rangle$ and $G_Y = \langle V_Y, E_Y \rangle$, respectively, and let $x, x' \in V_X$ and $y, y' \in V_Y$. Assume that $x \parallel y$ exists for all $x \in dom(X)$ and $y \in dom(Y)$. Then the representation $G_{X \parallel Y}$ of $X \parallel Y$ is given by the following equations:

$$V_{X \parallel Y} = \{x \parallel y \mid x \in V_X \wedge y \in V_Y\} \tag{6.8}$$

$$\begin{aligned} E_{X \parallel Y} = {} & \{\langle x \parallel y, x \parallel y' \rangle \mid x \in V_X \wedge \langle y, y' \rangle \in E_Y\} \\ & \cup \{\langle x \parallel y, x' \parallel y \rangle \mid y \in V_Y \wedge \langle x, x' \rangle \in E_X\} \end{aligned} \tag{6.9}$$

Recall that a graph $G = \langle V, E \rangle$ represents a preference list P if and only if $V = dom(P)$ and there exists a path in G from z to w if and only if $\langle z, w \rangle \in \mathcal{R}_P$. It is clear from equation 6.8 that $V_{X \parallel Y} = dom(X \parallel Y)$. Therefore, we must prove the following theorem in order to show that $G_{X \parallel Y}$ represents $X \parallel Y$.

Theorem 3 - There exists a path from $x \parallel y$ to $x' \parallel y'$ in $G_{X\|Y}$ if and only if $\langle x \parallel y, x' \parallel y' \rangle \in \mathcal{R}_{X\|Y}$.

Proof of $\Rightarrow$:

The proof uses induction on the length of a path. The base case consists of a path of length one. By equation 6.9, a path of length one consists of a single edge of either the form $\langle x \parallel y, x \parallel y' \rangle$, where $\langle y, y' \rangle \in E_Y$, or $\langle x \parallel y, x' \parallel y \rangle$, where $\langle x, x' \rangle \in E_X$. In the first case, $\langle y, y' \rangle \in \mathcal{R}_Y$ by equation 6.7, because G_Y represents Y. Therefore, $\langle x \parallel y, x \parallel y' \rangle \in \mathcal{R}_{X\|Y}$ by the definition of $X \| Y$ in equation 4.12. A similar reasoning holds for the second case.

For the induction step we assume that the theorem holds for the path $x_0 \parallel y_0, \ldots, x_n \parallel y_n$. If $x_0 \parallel y_0, \ldots, x_{n+1} \parallel y_{n+1}$ is a path in $G_{X\|Y}$, then by 6.9, the last edge in the path has form $\langle x_n \parallel y_n, x_{n+1} \parallel y_{n+1} \rangle$, where either $(x_n = x_{n+1} \wedge \langle y_n, y_{n+1} \rangle \in E_Y)$ or $(y_n = y_{n+1} \wedge \langle x_n, x_{n+1} \rangle \in E_X)$. Because G_X and G_Y represent X and Y, respectively, then either $\langle x_n, x_{n+1} \rangle \in \mathcal{R}_X$ or $\langle y_n, y_{n+1} \rangle \in \mathcal{R}_Y$. Therefore, by 4.12, $\langle x_n \parallel y_n, x_{n+1} \parallel y_{n+1} \rangle \in \mathcal{R}_{X\|Y}$. In addition, the induction hypothesis gives us that $\langle x_0 \parallel y_0, x_n \parallel y_n \rangle \in \mathcal{R}_{X\|Y}$. Because $\mathcal{R}$ is transitive, we have that $\langle x_0 \parallel y_0, x_{n+1} \parallel y_{n+1} \rangle \in \mathcal{R}_{X\|Y}$, which is the desired result.

Proof of $\Leftarrow$:

The proof follows by construction. Let $\langle x \parallel y, x' \parallel y' \rangle \in \mathcal{R}_{X\|Y}$. Then by equation 4.12, either $(x = x' \wedge \langle y, y' \rangle \in \mathcal{R}_Y)$, or $(\langle x, x' \rangle \in \mathcal{R}_x \wedge y = y')$, or $(\langle x, x' \rangle \in \mathcal{R}_X \wedge \langle y, y' \rangle \in \mathcal{R}_Y)$. In the first two cases the path consists of a single edge as given by either the left-hand or right-hand side of the union in equation 6.9. For the last case, $\langle x, x' \rangle \in \mathcal{R}_X$ means there is a path $x, x_1, \ldots, x_n, x'$ in G_X, and $\langle y, y' \rangle \in \mathcal{R}_Y$ means there is a path $y, y_1, \ldots, y_m, y$ in G_Y. Assume w.l.o.g. that $n \leq m$. Then, by definition, $G_{X\|Y}$ contains the path $x \| y, x_1 \| y, x_1 \| y_1, \ldots, x_n \| y_n, x' \| y_n, \ldots, x' \| y_m, x' \| y'$. Q.E.D.

A close examination of equation 6.9 gives some intuition as to how the graph of the result relates to the graphs of the operands. The set on the left-hand side of the union can be viewed as the union of an indexed set of graphs Y_{x_i}, where the vertices of Y_{x_i} are given by the set $\{x_i \parallel y \mid y \in V_Y\}$ and the edges by $\{\langle x_i \parallel y_j, x_i \parallel y_k \rangle \mid \langle y_j, y_k \rangle \in E_Y\}$. In other words, Y_{x_i} is just a copy of Y with each vertex y replaced by $x_i \| y$. The right-hand side of the union in equation 6.9 can be viewed as linking

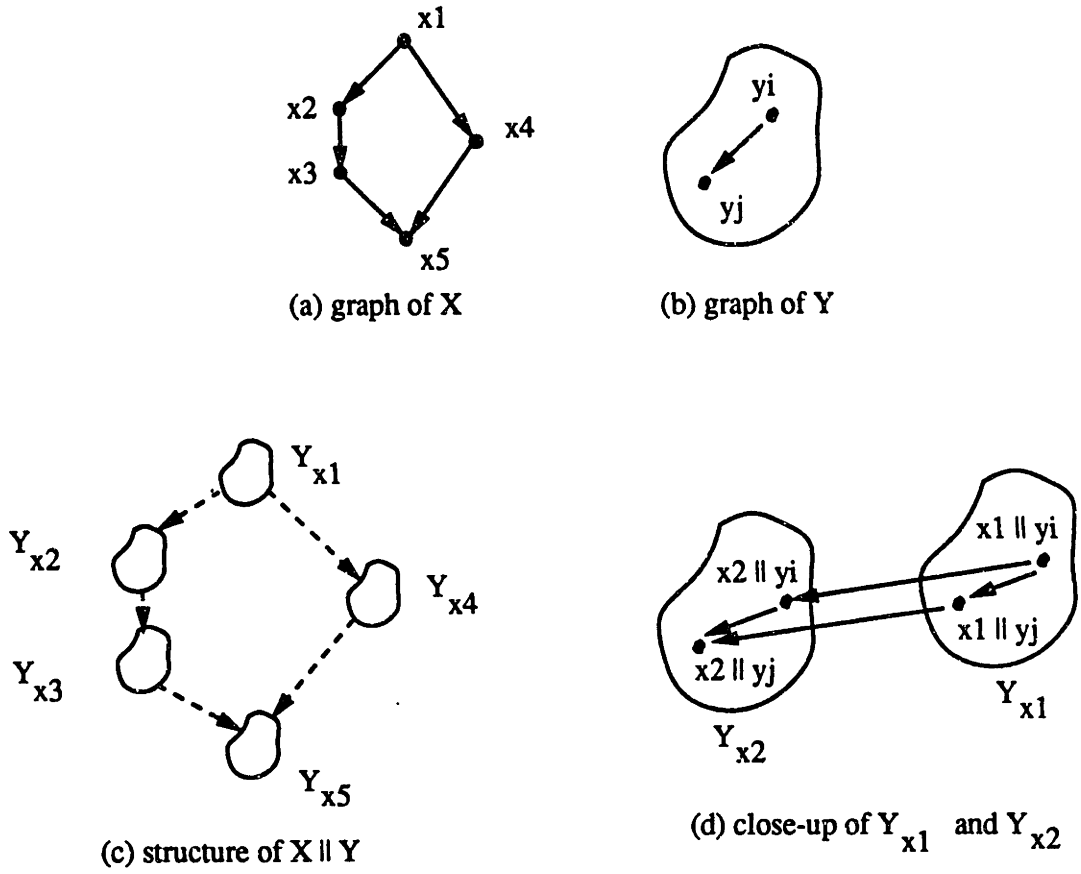


Figure 6-8: Example of consistent join graph structure

up corresponding vertices in the Y_{x_i}. For example, if $\langle x_i, x_j \rangle \in E_X$ then there is an edge from $x_i \parallel y$ to $x_j \parallel y$ for every $y \in V_Y$.

The intuition of copying one graph and linking corresponding vertices is illustrated in Figure 6-8. Part (a) in the figure shows an example graph for X where the x_i represent vertices and the arrows edges. Part (b) shows an outline of an example graph for Y with the detail for one edge $\langle y_i, y_j \rangle$. The exact structure of Y is not that important. Part (c) shows how one copy of Y is made for each vertex in X. The dashed arrows represent the links between corresponding vertices in the copies. Part (d) shows the detail of the edges in Y_{x_1} and Y_{x_2}.

The algorithm that implements the operation follows closely the definition given in equations 6.8 and 6.9. A high-level description of the algorithm is shown in Figure 6-9. The biggest problem is creating the edges because a vertex must already exist before an edge to that vertex can be created. The algorithm creates the vertices

```
consistent_join = proc(X:Graph, Y:Graph)
1   R:Assignment → Set[Assignment] := R[* ↦]
2   for x ∈ dom(X) and y ∈ dom(Y) do  R := R[(x || y) ↦ ∅]
3   for x ∈ dom(X) and y ∈ dom(Y) do
4       for y' ∈ Y(y) do  R := R[(x || y) ↦ (x || y') ∪ {R(x || y)}]
5       for x' ∈ X(x) do  R := R[(x || y) ↦ (x' || y) ∪ {R(x || y)}]
6   return R

note: roots(R) = {x || y | x ∈ roots(X) ∧ y ∈ roots(Y)}
```

Figure 6-9: Algorithm for consistent join

in lines 1 and 2, and initializes the adjacency list of each vertex to the empty set. The **for** loop in line 3 inserts all the edges. The nested **for** loop in line 4 creates all the edges in the copy Y_x of Y. The nested **for** loop in line 5 creates the edges between corresponding vertices of Y_x and $Y_{x'}$. On completion, the root set of the result consists of the join of the two operand's root-sets.

The efficiency of the algorithm depends on the implementation of the function $R(x \parallel y)$. The most efficient implementation uses an array for R and integer indices to represent the assignments. The vertices in X are numbered from x_1 to x_n, and those in Y from y_1 to y_m. The order in which the vertices are numbered is not important. The index of vertex $x_i \parallel y_j$ in R is given by the formula $m(i-1)+j$. Numbering the vertices in this way transforms the graph from a function with signature $Assignment \rightarrow Set[Assignment]$ to one with signature $Int \rightarrow Set[Int]$.

The above algorithm is valid only when all the assignments of one operand join with all those of the other. To handle the general case, we first use the above algorithm to construct the full graph as if all the assignments did join. If a vertex results from two assignments that did not join, then the algorithm simply marks the vertex and continues. After the graph is constructed, we remove the marked vertices in reverse topological order during a depth-first traversal of the graph (i.e., a vertex is processed only after all its successors have been processed).

The algorithm that removes the marked vertices is given by the procedure *purge* in Figure 6-10. Here, the above transformation of the graph from using assignments to integers is not only made explicit, but also required. The reason is that the join of

```
purge = proc(G:Int → Set[Int], marked:Set[Int])
1   dummy:Int := new_vertex(G)
2   G := G[dummy ↦ roots(G)]
3   marked := marked ∪ {dummy}
4   visited:Set[Int] := ∅
5   reachable:Set[Int] := ∅
6   G, visited, reachable :=
            purge_recursive(G, dummy, marked, visited, reachable)
7   return G[dummy ↦ ]

note: roots(G[dummy ↦ ]) = {v | v ∈ G(dummy) ∧ v ∉ reachable}

purge_recursive = proc(G:Int → Set[Int], v:Int, marked:Set[Int],
                       visited:Set[Int], reachable:Set[Int])
1   if v ∈ visited then return G, visited, reachable
2   visited := visited ∪ {v}
3   for s ∈ G(v) do
4       G, visited, reachable := purge_recursive(G, s, marked, visited, reachable)
5   for s ∈ G(v) do
6       if s ∈ marked then G := G[v ↦ (G(v) ∪ G(s) − {s})]
7   if v ∉ marked then
8       for s ∈ G(v) do reachable := reachable ∪ {s}
9   return G, visited, reachable
```

Figure 6-10: Algorithm to remove marked vertices from join

two assignments that disagree on the representation for a component is undefined. Therefore, any function that represents the graph cannot have the type *Assignment* as its domain because it cannot distinguish between different pairs of operand assignments that did not join. In the figure, the graph is represented by the argument G. The integers act as place holders for the corresponding pairs of assignments. The argument *marked* tells which assignments in the join are undefined.

The first three lines of the procedure *purge* set up the initial call to the procedure *purge_recursive*, which is the heart of the algorithm. Because of the way *purge_recursive* examines vertices, every vertex in the completely joined graph G must have an immediate predecessor. To ensure this condition, a dummy root is added to the graph in lines 1 and 2. The immediate successors of the dummy root are the original roots of G. The dummy root is removed before returning the result

in line 7.

The procedure *purge_recursive* traverses the graph in depth-first order, removing marked vertices as it goes. The argument *visited* keeps track of vertices that have already been processed. The argument *reachable* keeps track of all the vertices that are reachable from some other vertex that is not marked (i.e., the join of the assignments for this other vertex is defined). It is used to find the roots of the result as described below.

When called with a vertex v, *purge_recursive* first processes all of its successors (in line 4) before processing v. It then checks v's immediate successors to see if they are marked (line 6). When a marked successor s is encountered, it is deleted from the adjacency list of v, and s's immediate successors are inserted into v's adjacency list in place of s. If the visited vertex v is not marked, then its immediate successors are reachable, as indicated in line 7. Because the deletion and insertion are performed after all of v's successors have been processed, there are no marked vertices in the subgraph rooted at v with the exception of v itself. Therefore, the graph contains no marked vertices on completion except the dummy vertex. In addition, $v \in$ *reachable* for every vertex v that is reachable from some vertex in the result other than the dummy vertex.

It is possible to compute the root set of the result by looking only at *reachable*. However, this is much too expensive. Note that any root in the final result must be an immediate successor of the dummy vertex after the graph has been traversed. The reason is that all the predecessors of a root are marked and, therefore, the root propagates upward until it reaches the dummy. However, not all the immediate successors of the dummy vertex are in the root set. The root set can be computed efficiently by looking at the vertices in the dummy's adjacency list and checking if they are *reachable*.

The time required to compute the graph containing all the vertices is $O(v_1e_2 + v_2e_1)$, which is proportional to the number of edges in the result. The time required to remove vertices that did not join is $O((v_1v_2)^2)$.

Note that the algorithm above for removing vertices can introduce transitive edges. Even in the case where all vertices join, the result can have a transitive edge if one of the operands has a transitive edge. The following theorem states this more

strongly.

Theorem 4 - If G_X, G_Y, and $G_{X\|Y}$ are graphs representing preference lists X, Y, and $X \| Y$, respectively, and $x \| y$ exists for all $x \in dom(X)$ and $y \in dom(Y)$, then $G_{X\|Y}$ is a reduced graph if and only if both G_X and G_Y are reduced graphs.

Proof of $\Rightarrow$:

The proof is by contradiction. Assume w.l.o.g. that G_X is not a reduced graph. Then G_X contains a transitive edge. By definition, there exists a path $x_1, \ldots, x_n$ in G_X, where $n > 2$ and $\langle x_1, x_n \rangle \in E_X$. By equation 6.9, there exists a path $x_1 \| y, \ldots, x_n \| y$ in $G_{X\|Y}$, where $y \in V_Y$. In addition, $\langle x_1 \| y, x_n \| y \rangle \in E_{X\|Y}$. Because $\langle x_1 \| y, x_n \| y \rangle$ is a transitive edge, $G_{X\|Y}$ is not a reduced graph. A similar argument holds in the case that G_Y is not a reduced graph.

Proof of $\Leftarrow$:

It is actually easier to prove the contrapositive, which is stated in the following lemma:

Lemma 1 - Let $G_{X\|Y}$ be the graph of $X \| Y$ as described by equations 6.8 and 6.9. If the edge $\langle x \| y, x' \| y' \rangle \in E_{X\|Y}$ is a transitive edge of $G_{X\|Y}$ then either $\langle x, x' \rangle$ is a transitive edge of G_X or $\langle y, y' \rangle$ is a transitive edge of G_Y.

Sketch of proof:

First note that either $(x = x' \wedge \langle y, y' \rangle \in E_Y)$ or $(\langle x, x' \rangle \in E_X \wedge y = y')$ by equation 6.9. W.l.o.g., we consider only $(x = x' \wedge \langle y, y' \rangle \in E_Y)$. By hypothesis there exists a path $x_1 \| y_1, \ldots, x_n \| y_n$ in $G_{X\|Y}$ where $x \| y = x_1 \| y_1$, $x \| y' = x_n \| y_n$ and $n > 2$. First, it must be the case that $x_i = x_{i+1}$ for all $1 \leq i < n$ because otherwise, we would have that $\langle x, x \rangle \in \mathcal{R}_X$ by two applications of Theorem 3 and the definition of $X \| Y$. Therefore, for each pair of vertices along the path it must be the case that $\langle y_i, y_{i+1} \rangle \in E_Y$ by equation 6.9 and so $\langle y, y' \rangle$ is a transitive edge.

End of sketch.

Q.E.D.

6.4.4 Dominant Join

The algorithm to compute the dominant join directly from the operand graphs is similar to that for the consistent join in that the theorems and proofs are in essence alike. The main difference is the definition of the resultant graph. Therefore, I will give only a brief overview here.

Again we first assume that all the assignments in one operand join with all those in the other. The formal description of the result is as follows. Let X and Y be preference lists that are represented by the graphs $G_X = \langle V_X, E_X \rangle$ and $G_Y = \langle V_Y, E_Y \rangle$, respectively, and let $x, x' \in V_X$ and $y, y' \in V_Y$. Assume that $x \parallel y$ exists for all $x \in dom(X)$ and $y \in dom(Y)$. Then the representation $G_{X \gg Y}$ of $X \gg Y$ is given by the following equations:

$$V_{X \gg Y} = \{x \parallel y \mid x \in V_X \wedge y \in V_Y\} \tag{6.10}$$

$$\begin{aligned} E_{X \gg Y} = {} & \{\langle x \parallel y, x \parallel y' \rangle \mid x \in V_X \wedge \langle y, y' \rangle \in E_Y\} \\ & \cup \{\langle x \parallel y, x' \parallel y' \rangle \mid is_leaf(y, G_Y) \wedge is_root(y', G_Y) \\ & \wedge \langle x, x' \rangle \in E_X\} \end{aligned} \tag{6.11}$$

where $is_leaf(y, G_Y)$ is true if y is a leaf vertex of G_Y and $is_root(y, G_Y)$ is true if y is a root vertex of G_Y. The only difference from the definition of consistent join given in equation 6.9 is the right-hand side of the union.

The intuition of copying the Y graph, replacing each vertex y with $x_i \parallel y$ and linking the copies carries over. The difference is that the leaves of one copy are linked to the roots of another.

As for consistent join, in the case where not all assignments in one operand join with all those from the other, the graph is constructed as if they did join. Vertices that correspond to assignments that did not join are marked and then removed using the same algorithm as described for consistent join.

6.5 Implementation of Assignments

This section presents the specification of the assignment data type, describes its representation and explains the most pertinent implementation details. The specification is given in Figure 6-11, which provides a brief overview of the data type

and its most important operations. Assignments are mutable so that the parser can construct them efficiently while it is scanning an assignment expression. The only operation that mutates an assignment is *extend*. An assignment is no longer mutated once it is constructed. In contrast to most mutable data types, the identity of an assignment is not part of the abstraction. This is reflected in the operation *equal*, which states that two assignments are equal if they represent the same function.

There are two important factors that determine the structure of the representation. The first is that the most time-consuming operations are *join*, *differ*, and *equal*, which are used in the implementation of preference lists. The second is that assignments are used in hashed sets in the case where contexts are represented by assignments in a context list. These considerations lead to the following internal representation in a programming language like CLU:

```
assignment = struct[vcomps:array[vcomp_id],      % vspace component IDs
                    vreps:array[vrep_id],        % vspace representation IDs
                    hashval:memo]                % memoized hash value
memo = variant[undef:null, def:int]
```

The corresponding transfer representation would be similar, except it would not have the field *hashval*. The occurrences of **array** would be replaced by sequence32.

The most important representation invariant is that the array *vcomps* is sorted, which allows the join, differ and equal operations to be implemented efficiently. It also provides a canonical order for the components when computing the hash value of an assignment. This guarantees that two equal assignments have the same hash value.

If r is a value of type assignment, then the corresponding abstract value is given by the following equations:

$$sig(r) = \text{the elements of the array } r.\text{vcomps}$$
$$r(r.\text{vcomps}[i]) = r.\text{vreps}[i]$$

The first equation says that an assignment's signature consists of all the IDs found in the array vcomp. The second says that the representation ID for a given component is located at the same index as that component. Thus, vcomp and vrep are

assignment = **data type is new, extend, join, differ, hash, equal**

Overview

An assignment a is a function from vspace-component IDs to vspace-representation IDs. If u is a vspace-component ID in $sig(a)$, then $a(u)$ is the ID of the representation for u.

Operations

new = **proc**() **returns**(assignment)

effects: Creates a new assignment a with $sig(a) = \emptyset$

extend = **proc**(a:assignment, u:vcomp_id, r:vrep_id)
signals(duplicate, bad_vcomp_id, bad_vrep_id)

modifies: a

effects: Signals duplicate if u $\in$ sig(a). Signals bad_vcomp_id if u is not in the vspace database. Signals bad_vrep_id if r is either not in the vspace database or not a representation for u. Otherwise, extends a at u so that a(u) = r.

join = **proc**(x,y:assignment) **returns**(assignment)
signals(disagree(vcomp_id))

effects: Signals disagree if there exists a vcomp_id $\in$ (sig(x)$\cap$$sig$(y)) such that x(vcomp_id) $\neq$ y(vcomp_id). Otherwise, returns a new assignment that is the join of x and y.

differ = **proc**(x,y:assignment) **returns**(bool)

effects: Returns true if there exists a $u \in (sig(x) \cap sig(y))$ such that x(u) $\neq$ y(u). Returns false otherwise.

hash = **proc**(x:assignment, max:int) **returns**(int)

effects: Returns an integer value between 0 and max–1, inclusive.

equal = **proc**(x,y:assignment) **returns**(bool)

effects: Returns true if sig(x) = sig(y) and x(u) = y(u) for all u in sig(x). Returns false otherwise.

Figure 6-11: Specification of assignment data type

parallel arrays. The field *hashval* is used to memoize the assignment's hash value as explained below.

There are two additional representation invariants. The first is that if $u \in sig(r)$, then $r(u)$ denotes a valid vspace-component representation for u. The implementation of extend must verify this with the registry before extending an assignment. The second is that if that tag of *hashval* is *def*, then the memoized integer value of *hashval* is equal to the value of $hash(r)$, where $hash(r)$ is defined in equation 6.12 below.

The use of a sorted array is critical for the efficient implementation of the join, differ and equal operations. The implementation of join is similar to a merge sort algorithm. As the two assignments are merged the implementation checks to see that common vspace components have the same representation. Similarly, comparing two assignments for equality or difference involves scanning the two representations in order. Because the representation is sorted, the join, differ and equal operations run in time $O(n_1 + n_2)$ where n_i is the length of the i'th operand.

One additional trick that makes checking for equality fast under certain circumstances is to use pointer equality to "short circuit" the operation. In this case the implementation first checks to see if the two operands are the same object (i.e., have the same memory address). If the check fails then the assignments are compared as described above.

Assignments have a hash value so that they can be used in hash tables and hash-based sets. Because computing the hash value requires iterating over all the vspace component and representation identifiers, it is reasonably expensive. Therefore, the value, once computed, is memoized to avoid recomputation. This is the purpose of the field hashval in the representation. If the memoized hash value of r is n, then the operation hash(r,max) returns the value (n mod max). This requires that the memoized value n be suitable for a wide range of values max. The current implementation uses the following formula to compute the memoized value:

$$\sum_{i=1}^{high(r.vcomps)} r.vcomps[i] + primes[i] * r.vreps[i] \qquad (6.12)$$

where $high(r.vcomps)$ is the high index of the array $r.vcomps$, and $primes$ is a sequence consisting of consecutive prime numbers starting with 11 (i.e., 11, 13, 17,

...). The above formula works well for the cases examined in Section 6.7, where the IDs are relatively small integer values. The assumption of small integer values is reasonable for the representation IDs. However, a real system will likely have a different range of values for the component IDs and therefore, might require a different formula.

6.6 Overview of the Prototype Implementation

This section gives a brief overview of the prototype implementation. The implementation is written in CLU for two reasons. First, CLU is a heap-based language with automatic garbage collection, which makes it easy to implement algorithms that deal with complex data structures. Second, I am most familiar with CLU and, in particular, with CLUCC, which is a CLU-based LALR parser generator for use with CLU. These reasons made it possible for me to implement the system and demonstrate its feasibility in the shortest period of time. With a good CLU compiler the resulting system should not be much less efficient than an implementation in a "lower-level" language such as C.

The current implementation consists of a number of modules whose relationship is illustrated in the module dependency diagram shown in Figure 6-12. In the diagram, a box with a double line at the top represents a module consisting of an abstract data type. Plain boxes represent modules consisting of either a single procedure or a group of procedures that provide a single function, e.g., the "parser." An arrow means that one module depends on another in that it either uses the operations of the data type or invokes the specified procedure. The direction of an arrow indicates the direction of dependence. The diagram can be pretty much divided down the center into a parser subsystem on the left and a preference-/context-list subsystem on the right. The two subsystems share the pl-parse-tree and assignment modules.

The module "main" in Figure 6-12 represents a simple module that was used to measure the execution time for computing the best context given two preference lists. A brief outline of its implementation is given in Section 6.7, which describes how the execution times were measured.

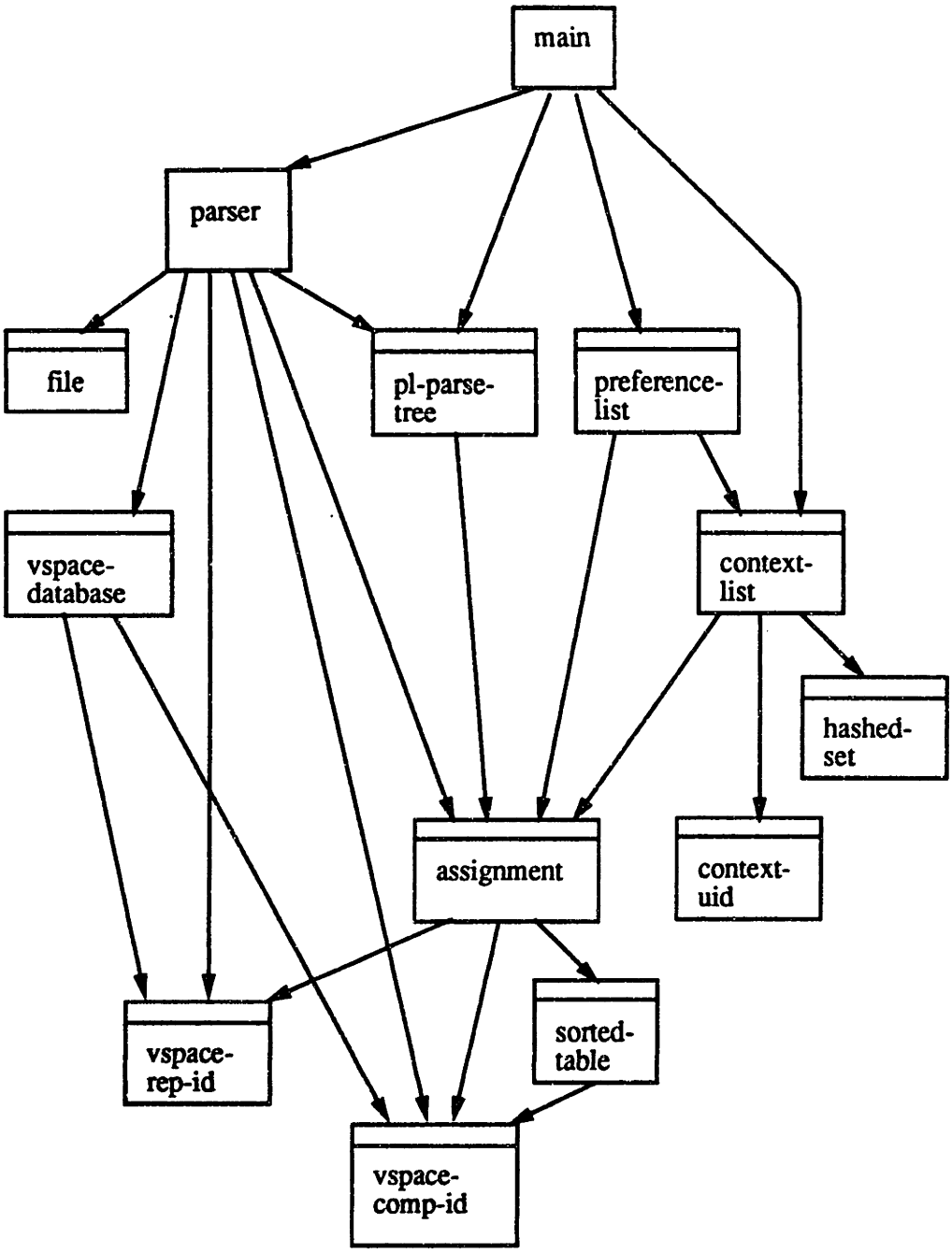


Figure 6-12: Module dependency diagram of implementation

The parser subsystem occupies the left half of the diagram and consists of the parser, file, and vspace-data-base modules. The parser module is actually a very large module consisting of an LALR parser (generated by CLUCC) and a number of semantic action routines. The parser converts a preference list expression in a file into a pl-parse-tree, which is an abstract representation of the expression. During the parse, the semantic routines convert each assignment expression directly into an instance of the assignment data type. This requires mapping the names of vspace components and representations to their unique identifiers. The mapping is supplied by the vspace-data-base module.

The vspace-data-base module in the current implementation is a trivial module that was cobbled together for the purpose of testing the rest of the implementation. In a real implementation the vspace-data-base module would communicate with the registry to define and look up vspace and negotiable-set definitions. Similarly, the vspace-comp-id and vspace-rep-id modules are special testing versions that represent vspace component and representation identifiers as simple integers. As discussed in Section 6.5, the exact representation of identifiers is not important as long as they can be ordered and manipulated as bit strings.

The preference-list subsystem occupies most of the right half of the diagram. The most important modules are the preference list, context list, and assignment data abstractions. Once a preference list is constructed from an expression, it is converted into one of the representations for a context list. For the case of an adjacency list the graph is reduced, while for the equivalence-class-list the graph is topologically sorted and the equivalence matrix computed. The current implementation does not check that every assignment is a context for the negotiable set because that requires a more extensive implementation of the vspace-database module than currently exists.

The context-list module provides the procedure for computing the best context. There are two different implementations of this module, corresponding to the two algorithms described in Section 6.2. The context-list module uses either the assignment module or the context-UID module, depending on how contexts are represented. In either case, it uses a hashed set to hold the set of supported contexts.

In a running application program, only the context-list module would be present, along with any modules it uses. The parser subsystem would be used only to

generate the context list, which would then be linked with the presentation layer as part of the application.

6.7 Measurements

The purpose of implementing the prototype and measuring its performance is to answer the following questions:

1. Which representation for context lists is better, an equivalence-class list or an adjacency list?

2. How does the time required to compute the best context compare to the savings in marshalling time obtained by using a more efficient representation?

To get a rough answer to these questions, I made several measurements of the time to compute the best context using the matrix, video display image, and VLSI vspace examples given in Chapters 4 and 5. The test bed used to make the measurements is outlined in Section 6.7.1 below. It is followed by a description of the examples in Section 6.7.2. The actual test cases and measurements are given in Section 6.7.3. This section closes with an analysis of the data.

6.7.1 The Test Bed

In all cases the programs were measured by compiling them with the PCLU compiler and using Pixie [DECa] to count the number of processor cycles consumed during execution. The PCLU compiler is a portable CLU compiler developed at M.I.T. that translates CLU programs into C. The resulting C programs were compiled using the GNU GCC[1] compiler with optimization level two. Pixie is a utility that instruments programs by dividing them into basic blocks and adding code to log the number of times each basic block is executed at runtime. The post-processing program Prof [DECb] uses the logged block-count information to calculate how many processor cycles were consumed by each procedure in the program as well as by the entire

[1]GCC is a widely available compiler from the Free Software Foundation.

```
(({float32:vax} > {float32:ieee}) ≫
 ({matrix_2d:colmaj} > {matrix_2d:rowmaj}))
||
(({int32:be} > {int32:le}) ≫ ({matrix_2d:colmaj} > {matrix_2d:rowmaj}))
```

Figure 6-13: Context list for matrix example

program. All test cases were run on a DECstation 5000/200,[2] which is a RISC-based CPU architecture with a cycle time of 25 MHz.

The measurements given in the following sections give the cycle count as reported by Pixie. The reader should keep in mind that the PCLU compiler produces code that runs approximately 2 times slower than code produced by a native CLU compiler.[3] The reason PCLU was used is that a native-mode compiler is not available for the DECstation, and Pixie on the DECstation provided the most reliable means to obtain accurate measurements.

6.7.2 The Example Context Lists

I used five example context lists to make the measurements in Section 6.7.3. The first example uses the context list in Figure 6-13, which is based on the matrix_2d vspace in Chapter 2. The next two examples are the context lists for the video display image and VLSI vspaces as given in Chapter 5. The last two examples are generated by joining context lists from the first three examples.

Some characteristics for each example are shown in Table 6.1. The column *name* gives the name used to refer to each example. In the case of *matrix, image*, and *vlsi*, the context list is found in the figure listed. The remaining context lists were obtained by joining the specified context lists. The table lists the number of contexts in each list, the number of edges in the case of an adjacency list representation, and the number of bytes required to store a context list using either the adjacency list or equivalence-class list representation.

The storage requirements were computed using equations 6.2 and 6.6 given ear-

[2]DECstation is a trademark of the Digital Equipment Corporation.
[3]This factor was obtained from Dorothy Curtis, the implementer of PCLU.

Name	Description	Num. of contexts	Num. of edges	Storage (bytes)	
				adj. list	e.c. list
matrix	Fig. 6-13, p. 142	8	13	128	136
image	Fig. 5-7, p. 90	8	9	112	136
vlsi	Fig. 5-18, p. 110	20	19	248	332
matrix-image	matrix‖image	64	168	1,196	1,160
image-vlsi	image‖vlsi	160	312	2,540	3,848

Table 6.1: Description of example cases

lier, with the value of c (the size of a context) set to zero. The reason for using $c = 0$ is that the most interesting measurement is the *difference* between the adjacency list and equivalence-class list representations. The storage required for the contexts is the same in both representations and, therefore, the difference is independent of a context's size. The reason the storage is computed rather than measured is that the prototype implementation of an equivalence-class list does not eliminate the storage for the zeros below the diagonal of the equivalence matrix. The computed value more accurately reflects what can be obtained. The values listed for adjacency lists, although computed, accurately reflect the storage used by the prototype implementation.

6.7.3 Test Cases and Data

The model for our tests is that one peer initiates negotiation by sending a context list to the other. The receiving peer then computes the best context and returns the result to the sender. There is one case that can be easily optimized, namely where both peers prefer the same context. In this case there is no need to compute *max_contexts*; the result is obtained by simply comparing the roots of the two lists. In a clever implementation the remainder of the received context list would not even be decoded. This case is not measured here because the cost is clearly on the order of a few tens of instructions, not including the overhead of receiving and dispatching the message.

All other cases require at least one application of *max_contexts*. If the result of the first application contains one context, then that context is the best context.

Otherwise, a second application of *max_contexts* is required. In the first application, the set S consists of the contexts in the context list of the peer with lower priority. When the receiver has priority, the sender's context list is used for S, in which case S must be constructed from the received data. On the other hand, if the sender has priority, then S belongs to the receiver and we can assume that it already exists; it was constructed either at compile-time or when the receiver was first initialized. Therefore, there are four different cases, depending on the number of applications of *max_contexts* and on whether the sender or receiver has priority. Given measurements for the time to compute *max_contexts* and construct a set of contexts, the time to compute *best_contexts* can be calculated for the different cases.

To measure the time for *max_contexts*, the test program read an example context-list from a file, parsed it, and then constructed the internal representation. It then read a second context list from which it constructed the set. The last step computed the result. A similar test program computed the time to create a hash set consisting of the contexts from an example context list. In both cases the overhead of reading and parsing the file was factored out by repeating the final computation many times and taking the average time as the result.

The time to compute *max_contexts*(C, S) is shown in Tables 6.2 through 6.4, which are described individually below. Each row in a table corresponds to the example context list C given by the name. Each row gives four measurements, corresponding to the four combinations of the representations for a context list ("adj. list" for adjacency list and "e.c. list" for equivalence-class list) and the representations for a context ("cuid" for context UID and "asnmnt" for assignment). All measurements are given in CPU cycles. The three tables correspond to different cases for the context list whose contexts constitute the set S. The tables correspond to the following three cases:

1. Best Case (Table 6.2). In this case, the set S contains the preferred context of C. Notice that the time is independent of the size of a context list in the case of a context UID. This is expected because the computation consists of looking up the sender's root context in the set S, and all context UIDs have the same size. In contrast, the size of an assignment depends on the number of components in the corresponding context.

2. Worst Case (Table 6.3). In this case, the *only* context in the context list C that is also in S is the least preferred context. Here the algorithm must traverse the entire context list.

3. Middle Case (Table 6.4). In this case, S contains a few contexts that are in the "middle" of the context list C. One interesting point is that for the equivalence class representation, the cost of this case is sometimes higher than the worst case given earlier. The reason is that there are no contexts in the equivalence class of the worst context. Therefore, the worst case only looks at the sequence of contexts. In the middle case it must also scan the equivalence class, which adds a significant factor.

A close look at the three sets of test cases above shows that the times required for both *matrix-image* and *image-vlsi* are dramatically larger than for the other example context lists. The importance of this observation is discussed in detail below.

The time required to construct a set consisting of the contexts for each example is given in Table 6.5. There are two entries for each example, corresponding to the two representations for a context.

Recall that the time to compute *best_contexts* depends on whether the sender or receiver has priority, and whether *max_contexts* is computed once or twice. If the sender has priority, then only the times for *max_contexts* are needed. If the receiver has priority, then the time for constructing the hashed context set must be added in. In the best case, *max_contexts* is computed once because the result contains one context. In the middle case, *max_contexts* must be computed a second time.

6.7.4 Analysis of Results

The beginning of this section poses two questions to be answered by measuring the prototype. In addition, Section 6.7.3 points out a dramatic increase in the time required to compute *max_contexts* in the *matrix-image* and *image-vlsi* examples, an issue that must be addressed. The reasons for the increased computation time are discussed first because they have the greatest influence on system design. The original questions are discussed afterwards.

Name	e.c. list cuid	adj. list cuid	e.c. list asnmnt	adj. list asnmnt
matrix	309	499	696	876
image	311	503	625	803
vlsi	310	501	779	958
matrix-image	314	509	874	1050
image-vlsi	320	522	974	1140

Table 6.2: Best case for *max_contexts*

Name	e.c. list cuid	adj. list cuid	e.c. list asnmnt	adj. list asnmnt
matrix	1040	2700	2470	4100
image	1040	2420	2310	3660
vlsi	2320	6150	4910	8560
matrix-image	6900	22700	18600	34200
image-vlsi	15300	49700	32800	66900

Table 6.3: Worst case for *max_contexts*

Name	e.c. list cuid	adj. list cuid	e.c. list asnmnt	adj. list asnmnt
matrix	1830	1870	3110	3120
image	840	1930	1810	2870
vlsi	1160	2880	3190	4880
matrix-image	10000	21100	19100	29900
image-vlsi	15600	32100	25000	41100

Table 6.4: Middle case for *max_contexts*

Name	Context UID	Assignment
matrix	2000	5200
image	2000	4400
vlsi	5000	15000
matrix-image	16000	54400
image-vlsi	36000	137000

Table 6.5: Time to construct a hashed context set

The dramatic increase in computation time is best illustrated by the *image*, *vlsi*, and *image-vlsi* measurements for equivalence-class lists using context UIDs from Table 6.4. In this case the sum of the *image* and *vlsi* measurements is 2,000 CPU cycles versus 15,600 for *image-vlsi*. Adding in the estimated set construction times from Table 6.5 for the case of receiver priority gives 9,000 versus 51,600. In both cases the difference is nearly an order of magnitude.

There are two main reasons for the increase, both of which are related to the property that the *image* and *vlsi* vspaces are "independent," i.e., they do not share any vspace components. First, the number of contexts in the join of two preference lists for independent vspaces is the product of the number in the operands. Second, the result contains a large number of unordered contexts. The result is that the time to compute the best context for a negotiable set containing many independent vspaces is substantially larger than if the set were divided into subsets and each subset computed separately.

Given the current design, a system designer must be careful how the vspaces in a large "interface" are organized into negotiable sets. A poor decomposition can result in a high negotiation cost. Note that negotiating several sets separately need not introduce extra message overhead because the individual sets can be packaged up into a single message.

The data suggest that it is worth considering a different definition for a context than that in Chapter 3. It might be better to organize the structure of a context around vspaces instead of vspace components, e.g., by specifying preferences for each vspace independently of any other vspace. However, it is not clear *a priori* that this is the right approach. One problem that remains is separating the vspaces that are important from those that are not. For example, in an interface for handling VLSI masks, there might be many unimportant vspaces that share components. Negotiating preferences for each vspace individually could introduce more overhead than if there were one set of preferences for all of them.

It is not possible to draw a definite conclusion at this point because it is not clear if the problems exhibited in the data are typical of "real" interfaces. Here I have arbitrarily taken two very different vspaces that have several components each, and joined them to form a negotiable set. Whether this is realistic is debatable. In

addition, there are other design issues involving contexts that are discussed later in Chapter 9. Any conclusions must wait until then.

We are now ready to look at the original questions, the first of which is: which representation is better, the adjacency list or the equivalence-class list? Judging by the figures in Tables 6.2 through 6.4, the equivalence-class list is about 2 to 3 times faster than the adjacency list. However, as context lists get larger, the equivalence-class list requires more storage. The reason is that the storage required by the prototype implementation is $O(n^2)$, where n is the number of contexts. An adjacency list requires $O(n + e)$, where e is the number of edges. Because "larger" context lists are usually constructed by joining "smaller" ones, it is generally the case that $O(e) < O(n^2)$. Table 6.1 shows that there is already a marked difference in the *vlsi* test case and a substantial difference in the *image-vlsi* test case. Therefore, the two representations provide a space-time tradeoff, with equivalence-class lists requiring less time and adjacency lists less space.

As mentioned in Section 6.2.2.1, the bit-vectors in an equivalence class can be represented using run-length encoding. Although the worst-case storage required is $O(n^2)$, the average case is probably much better. An interesting topic for future research is how it compares on average to the other representations, for both storage and time.

The second question is: how does the time required to compute the best context list compare to the savings in marshalling time? To answer this, we only need examine the case of integer matrices to see that negotiation provides a positive benefit. As shown in Chapter 1, a 100-by-100 integer matrix requires approximately 70,000 CPU cycles just to transpose the matrix. Assuming the worst-case time with receiver priority and assignments, the cost to negotiate for matrices is about 5,000 to 10,000 cycles. The benefit is realized with the first 100-by-100 matrix. Even if we assume the worst case time with receiver priority for the combination of matrices and images, the time is about 60,000 cycles. If we throw in the time to process the negotiation message and decode the context list, negotiation brings a positive benefit with just a few 100-by-100 matrices.

An alternative characterization is obtained by comparing the cost of negotiation with that of a null RPC (i.e., invoking a remote procedure with no arguments

and receiving a reply message with no results). An optimized null RPC requires about 2.5 milliseconds of CPU time on the Firefly, which uses a one MIP micro-VAX II processor [SB89]. The corresponding figure for Amoeba, which is a 1.5 MIPS Motorola 68020, is about 1.5 milliseconds [vRvST89]. Direct comparison is somewhat difficult because these figures are given in milliseconds instead of CPU cycles, and the processor architectures are very different. If we assume a very liberal figure of 2,000 CPU cycles for a null RPC and use the maximum figures for negotiation, then the cost of negotiation ranges between 1 and 50 RPCs. If we use the figures for equivalence-class lists with context UIDs and assume a reasonable decomposition of the vspaces into negotiable sets, then the cost is more on the order of 1 to 5 null RPCs.

In summary, negotiation is practical, and the cost of negotiation is worth the gain in marshalling efficiency with data sets that are on the order of a few thousand bytes. The break-even point is likely to be even lower in the common case where applications are designed to work together and therefore, either prefer the same context or support the context preferred by their peers.

Chapter 7

Negotiation in Real Systems

Negotiation is the process by which two peers agree on a context for a negotiable set and a context identifier (CID) that is used to denote the agreed context in messages. Previous chapters have discussed the concepts of contexts, negotiable sets and context lists, as well as algorithms for computing the best context from two context lists. This chapter shows how these concepts and algorithms could be employed in Mercury, OSI, and OSF's RPC facility. Each system is described in its own section.

7.1 Negotiation in Mercury

This section does not define a negotiation protocol for Mercury. Rather, it answers the following questions:

1. What are the negotiable sets?
2. How do programmers specify context lists?
3. When does negotiation occur?
4. What information is exchanged during negotiation?
5. How are CIDs used?

Given answers to these questions, it is relatively straightforward to design the low-level details of exchanging context lists and assigning CIDs.

Negotiable sets are associated with Mercury interface definitions, which are described first in Section 7.1.1 below. The precise definition of the negotiable set associated with a given interface is then given in Section 7.1.2. As will be seen, it is necessary to restrict the binding of interfaces to ensure that all parties use the same negotiable set. These restrictions are given in Section 7.1.3. This is followed by a discussion of how context lists are specified in Section 7.1.4.

Sections 7.1.5 through 7.1.7 answer the remaining questions. Section 7.1.5 describes when to negotiate. It also describes the use of "canonical" contexts to guarantee connectivity and avoid the need to negotiate before sending data. Section 7.1.6 describes the information exchanged during negotiation. Section 7.1.7 discusses some fine points about the use of CIDs and the issue of synchronizing negotiation with data transfer.

Any design for negotiation must make tradeoffs among efficiency of negotiation, ease of use, flexibility of binding and overhead of synchronization. In this section I discuss in detail a particular set of design choices and how these choices affect individual tradeoffs. The section concludes with a summary of the tradeoffs.

7.1.1 Mercury Ports and Interfaces

A *port* is a procedure-like object that can be invoked by a Mercury peer entity. In Mercury, a peer entity is referred to as a *module*. Each port is associated with a specific module. Some ports are created when a module first comes into existence, while others can be created dynamically.

An *interface* is a group of ports. In the simplest case, all the ports in a given interface are associated with the same module. This configuration is referred to as a *central server*. It is the only possible configuration in many commonly used systems [FS91, DLM+87, Sun85, ISO88b]. In Mercury, individual ports in a given interface may be associated with different modules, a configuration that is referred to as a *distributed server*. In this case a set of modules collectively provides a single service. From the client's point of view, the server modules together act as if they were a single server. With respect to negotiation, the client must negotiate individually with each server module, but this is usually handled by the Mercury

presentation layer without involving the application-level code. In most cases, the application-level code of a client need not be aware that a service is distributed.

The terms client and server are actually relative terms. A module acts as a *client* of an interface if it invokes any port in that interface. Similarly, a module acts as a *server* if it provides any port in an interface.

Both ports and interfaces are defined using an interface description language. Here we extend the language Amalgam introduced in Chapter 2 to incorporate ports and interfaces. A port definition has the form:

id = **port**(*Vspace-exp*,...) **returns**(*Vspace-exp*,...)
signals(id$_1$(*Vspace-exp*,...), ...)

The port's arguments are given by the list of vspaces following the keyword **port**, and the normal return values are given by the list following the keyword **returns**. There may be zero or more arguments and return values. The **signals** clause gives a list of named exceptions that can be returned in lieu of a normal result. Each exception may be accompanied by zero or more values.

An interface definition has the form:

id = **interface**
port definition 1
⋮
port definition n
end

where each port definition has the form given above.

Ports and interfaces are both vspace constructors. Although port and interface definitions denote vspaces, they are not vspace expressions, i.e., they are not productions of the non-terminal *Vspace-exp* in the grammar given in Figure 2-3 on page 42.[1] Instead, they introduce new identifiers that may appear in vspace expressions, including expressions that appear as an argument, result, or exception in a port definition. Therefore, port and interface values are first-class values.

[1]The main reason they are not expressions is that some languages do not support anonymous procedures. Making them expressions would make it difficult for stub compilers to generate a named procedure to represent a port in such a language.

7.1.2 Negotiable Sets

Deciding what vspaces constitute a negotiable set requires making some tradeoffs. As will be seen in Section 7.1.7, all of the "application-level" vspaces in a single invocation or response message are encoded using the same context. Therefore, the smallest negotiable set would consist of the set of vspaces in either an argument list, return list, or exception list for a single port. At the other end, the largest set of vspaces is the set containing all the vspaces that occur in an interface. The only other alternatives are: 1) use the set consisting of all the vspaces in a port's argument list, return list, and exception list, or 2) require the programmer to specify the negotiable sets as part of an interface definition.

The two main factors that affect the choice of a negotiable set are efficiency of negotiation and economy of expression. If an interface has many ports and each port constitutes a negotiable set, then the overhead of negotiating for each individual port is higher than negotiating once for the entire interface. In addition, it would be rather tedious to require programmers to specify a context list for each port. The simplest choice is to define a single negotiable set for each interface.

However, the simplest choice may not be the most efficient choice. As discussed in Chapter 6, negotiation can be costly if a negotiable set contains several independent vspaces, where each vspace consists of a number of components. In such cases it is better to partition the negotiable set into smaller negotiable sets. The approach used here is to define a single default negotiable set and allow the interface designer to override the default by explicitly specifying the negotiable sets. The default negotiable set is defined below, followed by a discussion of how one would partition it into multiple negotiable sets.

The default negotiable set of an interface definition I is given by the function *neg_set*(I), which is defined as follows:

$$neg_set(I) = \{p \mid p \text{ is a port defined in } I\} \tag{7.1}$$

When an interface definition is entered in the registry, its default negotiable set is assigned a UID.

The above definition, although simple, does not quite tell the whole story. When specifying a context for a negotiable set, one must specify representations for the

components of all the vspaces in the set, which in this case are ports. Therefore, we extend the function *components* defined in Section 3.1.1 as follows:

$$components(id) = \begin{cases} \emptyset, & \text{if } id \text{ denotes an interface} \\ \bigcup_{v \in vspaces(p)} components(v), & \text{if } id \text{ denotes a port } p \end{cases}$$

where $vspaces(p) = \{v \mid v \text{ appears as an argument, result, or exception in } p\}$ (7.2)

The reason for not including any components from an interface in the above definition is that negotiation takes place with respect to a single interface. Any interface that is obtained via a port invocation is considered distinct from the interface that contains that port. Intuitively, a port p denoted by identifier *id* "is contained in" interface I if it is defined in I or its identifier *id* is encountered during the computation of the components of I. A formal definition of the relation "is contained in" is omitted because it would be very similar to that for *components*. By definition then, any port that is obtained via a series of one or more port invocations starting with a port defined in I is contained in I.

The above definitions of *neg_set* and *components* make it clear that a negotiable set includes only vspaces of interest to the application. It does not include the Mercury protocol headers that precede the application data in the message; nor does it include the vspaces for ports and interfaces. The reason is that application programmers specify context lists only for application vspaces and any aspect of the underlying Mercury protocol should be invisible to the application. The transfer representations of the protocol headers and the descriptors for ports and interfaces are considered to be a part of the Mercury protocol implementation. The design of an interface description language that allows complex linguistic constructs like ports and interfaces to be defined in such a way that the protocol-level vspaces can be defined and negotiated separately from the application-level vspaces is an issue for future research.

I now consider the case where negotiation can be made more efficient by partitioning the default negotiable set. Recall from Section 6.7.4 that the need to partition the set is a result of the current definition of a context. In the ideal case,

the problem would be solved by finding a different definition for context that made negotiation efficient, regardless of the number of vspaces in the interface. Then the default negotiable set would be the only negotiable set. This would remove the burden of having to partition the interface, thereby keeping the interface specification simple.

If the current definition of a context is used, then the language construct from Section 7.1.1 for defining an interface must be extended so that the interface designer can specify a collection of negotiable sets for an interface. The actual syntax is not that important. The only requirement is that the collection cover the default negotiable set. The simplest way to do this is to divide up the ports in the interface into individual sets such that a given port is in exactly one set.

7.1.3 The Binding Model

Ports and interfaces are first-class values in that they can be passed as part of an argument or result of a port invocation. Typically, an interface is created by a single module and then *exported* to the Mercury interface registry. A client wishing access to a service can look up and *import* the interface from the registry. The client then notifies the server that it intends to use the interface. The process of exporting, importing and notifying the server is referred to as *binding.*

When a client invokes a server port it may pass another port or interface as an argument or receive one as a result. In addition, a module can decompose an imported interface into individual ports and create new interfaces using any combination of ports in its possession. The ability to create ports dynamically, pass them among Mercury peers and group them into interfaces provides a powerful mechanism for creating distributed programs.

The first-class status of ports and interfaces and the ability to construct interfaces at run-time can lead to problems with negotiation. One such problem is illustrated by the example shown in Figure 7-1. In the figure, module C exports port p_1 as part of interface I_1, which is then imported by module B (e.g., via the registry). B then incorporates p_1, along with some of its own ports, into interface I_2, which is then imported by A. From A's point of view p_1 belongs to I_2. Furthermore, A is not even

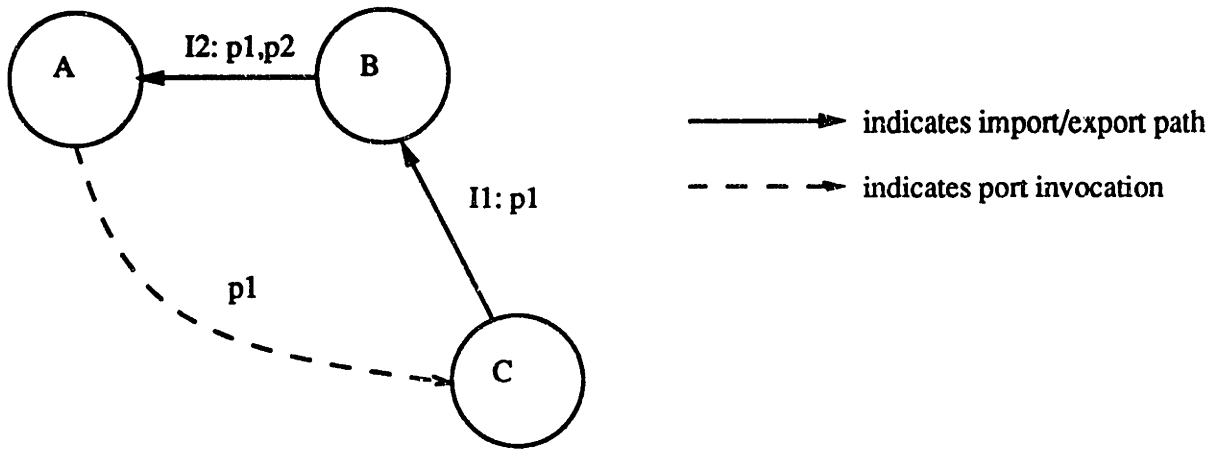


Figure 7-1: Problem with unrestricted interfaces

aware of interface I_1. Therefore, when A receives p_1 it will negotiate with C for the negotiable set associated with interface I_2. Even if p_1 were not incorporated into I_2, a similar problem would occur if B returned p_1 as the result of A's invoking p_2. The only way to ensure that negotiation succeeds is to require that C be cognizant of I_2.

Therefore, in order for negotiation to make sense we must place restrictions on the run-time mechanisms that construct interfaces and pass ports as arguments and results. The restriction is that a port may be included in an interface I or passed via another port contained in I only if the port's owner is able to negotiate for interface I. In order to implement this restriction, a port's abstract representation includes the set of interface UIDs for which the port's owner can negotiate. This allows the Mercury run-time system to check the restriction. In addition, clients can check that an imported interface was correctly constructed before initiating negotiation, if they so desire. This restriction is not a problem for a distributed server because the server modules are designed to work together and are cognizant of the interface provided to the client.

Restricting the binding of interfaces is also important for efficient negotiation. From Chapter 6 we know that negotiation is most efficient when we negotiate for a set of vspaces using context UIDs. Because context UIDs are relative to a negotiable set and negotiable sets are associated with interfaces, two modules can negotiate using context UIDs only if they negotiate for the same interface.

7.1.4 Specifying Context Lists

Programmers specify the context list for a module's use of an interface using an *annotation language* that incorporates the preference expression language given in Chapter 4. Annotations are kept separate from the interface definition and do not reside in the registry. A "stub compiler" takes an interface definition and annotations, and generates the run-time data structures needed by both the application and the Mercury presentation layer. Separating the annotation language from the interface definition language allows the annotations to be tailored to a specific programming language.

Specifying a context list for a module that uses every port contained in an interface poses no problems other than figuring out how to express it. On the other hand, when not every port is used, there may be vspaces in the negotiable set that are never communicated. This can occur, for example, in a distributed server when one of the server modules provides only a part of the service. It can also occur when a client uses only a subset of a service. These cases pose the additional problem of what to do with the unused vspaces. Requiring programmers to specify preferences for them is burdensome. However, if no preferences are specified for them, then the contexts in the context list will not be complete for the negotiable set.

This problem is the result of associating negotiable sets with interfaces and illustrates another tradeoff in choosing the size of a negotiable set. If negotiable sets were associated with the individual ports in an interface, then this problem would not occur. As pointed out earlier, requiring a programmer to specify a context list for each port can be quite burdensome, especially if there are many ports. The solutions given below let programmers specify a context list for a subset of an interface, while still negotiating for the entire interface.

There are two solutions, depending on whether a context is represented by an assignment or a context UID. Both solutions are based on the following observation. Let C be a context list for the negotiable set N associated with an interface. Also, let C_A be a context list for some $N_A \subseteq N$ as specified by module A, and C_B be a context list for some $N_B \subseteq N$ as specified by module B. The key observation is that any vspace that is not in $N_A \cap N_B$ is never transmitted between modules A

and B. Therefore, we only need consider the common vspaces when computing the best context.

If contexts are represented by assignments, then a context from C_A *is similar to* a context from C_B if both contexts assign the same representation to the vspace components that they have in common. Similarity is used to define set membership, which is then used to define *max_contexts*. The following definitions make this precise:

$$\begin{aligned} x \sim y &\Leftrightarrow \forall u \in (sig(x) \cap sig(y)) \,.\, x(u) = y(u) \\ x \,\tilde{\in}\, S &\Leftrightarrow \exists y \in S \text{ s.t. } x \sim y \\ max_contexts(C,S) &= \{c \mid c \in dom(C) \wedge c \,\tilde{\in}\, S \wedge \nexists c' \,\tilde{\in}\, S \text{ s.t. } \langle c', c\rangle \in \mathcal{R}_C\} \end{aligned}$$

The main difference between the definition of *max_contexts* above and equation 3.5 on page 63 is in the term $c \,\tilde{\in}\, S$. Recall from Chapter 6 that a fast implementation for set membership is important for computing the best context. For this reason, sets are implemented as hash tables. However, in this case, we cannot compute a hash function because different context lists support different subsets of the interface and, therefore, there is no common set of vspace components. The result is that computing the best context is much less efficient.

The solution for context UIDs takes a slightly different tack. The problem here is that we cannot check if two context UIDs specify a common representation for some subset of vspace components; we can only compare context UIDs for equality. The key point in this case is that a context that is complete for N can be substituted for a context in C_A or in C_B as long as it specifies the same representations for the vspaces in $N_A \cap N_B$. The solution is to have the stub compiler "extend" the specified context list so that it uses contexts that are complete for N.

The extension of a context list is defined as follows. Let I be an interface, P the set of ports contained in I, $P' \subset P$ the set of ports actually used, and C' a context list for P'. First, we define the function *ext_asn*, which takes an assignment a and an interface I, and computes the set of extensions of a that satisfies the following formula:

$$ext_asn(a, I) = \{c \mid is_context(c, neg_set(I)) \wedge (c \mid sig(a) = a)\}$$

Computing the extension requires adding mappings for all possible representations for all the vspace components in $neg_set(I)$ that are not in P'. The representations for a given component are obtained from the registry. The stub compiler then computes all possible closures for each such vspace component that is an abstract vspace or constructor. The context list C for I that is the *extension* of C' is defined as follows:

$$dom(C) = \bigcup_{a \in dom(C')} ext_asn(a, I) \tag{7.3}$$

$$\langle x, y \rangle \in \mathcal{R}_C \Leftrightarrow \exists\, x', y' \in dom(C) \text{ s.t. } (x \in ext(x', I) \wedge y \in ext(y', I) \wedge \langle x', y' \rangle \in \mathcal{R}_C) \tag{7.4}$$

Intuitively, C is computed by replacing each context a in the partial order of C' with an equivalence class consisting of the extension of a. Because each extension of a has a corresponding context UID, the resulting context list can be used to negotiate for the interface.

With the above solution, the computation for *max_contexts* remains the same. The problem is that an extended context list can be very large, perhaps to the point of being impractical.

In summary, there are three possible solutions discussed above: 1) require the programmer to specify a negotiable set for each port, 2) use assignments to represent the specified context list, but use a different definition for *max_contexts*, and 3) use context UIDs and extend the specified context list. Each solution has advantages and disadvantages. The best solution is a question for future research.

7.1.5 When to Negotiate

There are two kinds of contexts in Mercury: canonical and negotiated. A *canonical context* for an interface is a specially designated context that every module that uses the interface agrees to accept. Canonical contexts provide the following advantages:

1. *Connectivity.* – Communication between all modules using the interface is guaranteed.

2. *Low overhead for "short" sessions.* – In the case where the total amount of data sent during a session is small, the cost of negotiation can be greater

than the benefit of using an optimal context. Overhead is reduced by not negotiating and using a canonical context instead.

3. *No round-trip synchronization before transferring data.* – A canonical context can be used to transfer data without negotiating or before negotiation is complete. This allows data transmission to occur in parallel with negotiation. This is in contrast to OSI's connection-oriented presentation protocol.

There may be more than one canonical context for a given interface. The reason for permitting more than one is that it allows a system to be optimized for common patterns of communication. Systems are often composed of a few sets of components, where the components within a given set are designed to work together and, therefore, prefer the same context. When a module initiates communication with a peer, it typically sends the data using the canonical context that it prefers most. There is usually no need to negotiate if the peer belongs to the same set.

The canonical context definitions are stored in the Mercury registry along with the interface definition. The registry assigns every canonical context a UID, which also can be used as a CID in messages.

Points 2 and 3 above make it clear that negotiation is not needed to communicate; rather, it is needed only for efficient communication. The question as to when to negotiate is, therefore, an application-dependent issue. All that is required is that the presentation layer implementation provide the application with a service interface that allows the application to decide when to negotiate. The implementation may also provide a default behavior. For example, because negotiation typically costs on the order of a few round-trip port invocations, the presentation layer may automatically initiate negotiation after some small number of invocations.

7.1.6 What to Negotiate

This section describes the information exchanged during negotiation. In all cases the programmer specifies a context list for the negotiable set associated with the interface as described in Section 7.1.4. The stub compiler generates an equivalence-class list representation with context UIDs as described in Chapter 6. The context UIDs are retrieved from the registry at compile time.

When one module initiates negotiation, it sends the following information to its peer: 1) the UID of the negotiable set, 2) an encoding of the equivalence class representation, and 3) its priority. The responding module computes the best context as outlined in Chapter 6 and returns the corresponding context UID as the CID to be used in messages.

Using context UIDs in the representation for a context list results in the most efficient negotiation as well as the most compact representation. In addition, as will be seen in the next section, using a context UID as the CID also eliminates the need to synchronize negotiation with the use of the CID in messages.

7.1.7 Using CIDs

CIDs are used in port invocation and response messages to denote the context used for encoding the application-level data (i.e., argument, return, or exception values). The CID is located in the Mercury protocol header, which precedes the application data. As it currently stands, Amalgam cannot be used to define the protocol headers. It not only lacks low-level data types like bit-vectors, but it also does not allow CIDs to be part of a vspace definition.

The use of a CID in the data stream must be synchronized with its definition. Otherwise, the message's recipient would not be able to decode the data when it arrives. One of the primary advantages of using context UIDs as CIDs is that there is no need for run-time synchronization because they are defined at compile time. Once a module has computed the best context during negotiation, it can immediately use its UID in a message. If CIDs were assigned during negotiation then a CID's definition must arrive at the recipient before any messages that use it. Synchronizing assigned CIDs is not a problem if the underlying session and transport layers provide ordered delivery and negotiation messages can be sent in-band with the data.

Mercury, on the other hand, allows modules to have concurrently executing agents, where each agent has a logically separate session[2] with a peer module [Lis87a]. In addition, two sessions with the same peer are not required to share

[2]In Mercury terminology these sessions are referred to as "call streams."

the same transport connection. It is possible to negotiate a context for all agents that are connected to the same peer and that use the same interface. This avoids having to negotiate individually for each session. Because sessions do not necessarily share a transport connection, using context UIDs avoids having to synchronize each session individually.

7.1.8 Summary of Design Tradeoffs

The design presented above for negotiation in Mercury makes a number of choices that involve tradeoffs among efficiency of negotiation, ease of use, flexibility of binding, and overhead of synchronization. The following list gives the primary design decisions that were made.

1. Negotiable sets are associated with interfaces.
2. Programmers specify a context list for the subset of the interface that is actually used by a module. The "stub compiler" then extends the context list to cover the entire interface.
3. Canonical contexts are used to achieve connectivity.
4. Negotiated contexts are used to achieve efficient communication.
5. The representation of a context list uses context UIDs.
6. Context UIDs are used as CIDs.
7. A port may be included in an interface only if the module that created the port is cognizant of that interface.

Of the above decisions, the first is the most important because it directly affects the tradeoffs between efficiency of negotiation and ease of use. Negotiating for an interface generally requires less overhead than negotiating for each individual port. In addition, specifying a single context list for an interface is usually less tedious than specifying one for each port or individual argument, return, and exception list.

The last three decisions above involve the tradeoffs among efficiency of negotiation, overhead of synchronization, and flexibility of binding. Using context UIDs

in the representation of a context list results in the most efficient negotiation. Using context UIDs as CIDs avoids the round-trip communication needed to assign a CID and eliminates synchronizing the definition of a CID with its use in messages. However, it requires the restriction on binding that a port can be included in an interface only if that port's owner can negotiate for that interface. In the worst case, lifting this restriction could result in modules that cannot communicate. The reason is that canonical contexts are defined with respect to an interface. If the owner of a port does not recognize the interface, then the user of that port cannot assume that it has a context in common with the owner. Even if it did have a context in common, it would not be able to negotiate because the contexts in a context list are identified by context UIDs, which are relative to the negotiable set associated with the interface.

7.2 Negotiation in OSI

This section describes briefly how negotiation with context lists could be added to OSI. It assumes that the reader has some familiarity with the OSI presentation layer and the remote operations service. The description is in two parts. The first part outlines the steps required to add context lists to the connection-oriented presentation layer in general. The connection-less mode of operation is not considered because it does not provide any facilities for negotiating a transfer syntax. The second part addresses the specific case of using context lists for remote operations (ROS). In both cases the goal is to minimize the amount of change needed while obtaining the advantage of using context lists.

7.2.1 OSI in General

The presentation layer model and protocol, as specified in [ISO88d] and [ISO88e], define a general framework that is independent of the use of ASN.1. Therefore, the discussion below is divided into two parts. The first describes changes to the presentation layer model and protocol, while the second describes changes to ASN.1 so that it fits the new model.

7.2.1.1 Changes to Presentation Model

The following steps show how to add negotiation with context lists to the OSI presentation layer model, independently of the abstract syntax notation.

1. *Make negotiable set synonymous with named abstract syntax.* This is the most logical association for a negotiable set because a named abstract syntax already identifies a set of types and the presentation layer currently negotiates a transfer syntax for each named abstract syntax individually. This association also minimizes the change to the current presentation layer model.

2. *Make context synonymous with transfer syntax.* Again, this is the most logical choice because a context acts as a set of "encoding rules" that define a transfer syntax for a given abstract syntax.

3. *Make context UID synonymous with named transfer syntax.* A named transfer syntax is denoted by an *object identifier* (OID), which is a unique name assigned by an authority designated by a standards process [ISO87a, Annex B]. Therefore, OIDs can be used as context UIDs.

   The remaining changes to the semantics of the presentation protocol would be minor if we could use OIDs for every context. In this case the presentation protocol machine[3] could compute the best context without interpreting an OID, thereby avoiding the need to introduce the machinery of assignments for describing contexts. However, a given negotiable set is likely to have a large number of possible contexts. In order to use OIDs as context UIDs in such a case, an automated authority must be set up to assign the OIDs, much like the Mercury registry assigns context UIDs. Without an automated authority, OIDs must be assigned by some authorized person or body. In this case, it is unlikely that an OID can be assigned to every context. In any case, it is necessary to assign OIDs to some contexts for use as default or canonical contexts as outlined under changes to the protocol in item 7 below.

[3]The presentation protocol machine is that part of the presentation layer that actually carries out the protocol and interprets protocol messages. Its current definition is given in [ISO88e].

4. *Introduce assignments as "transfer syntax descriptions."* As argued above, OIDs cannot be used in general to represent every context. Therefore, in a presentation protocol message containing a context list, the contexts can be represented as assignments. This requires the introduction of the concepts component, abstract representation, transfer representation, assignment, and context. It is possible to formalize the material in Sections 3.1.1 and 3.1 so that the concepts are independent of the notation used to define them.

   Note that a formal definition of the concepts is only required in order to define precisely which assignments are valid contexts for a given named abstract syntax. It is not needed by the presentation protocol machine to compute the best context, because the algorithms in Chapter 6 do not interpret the identifiers that denote components and representations.

5. *Assign identifiers to components.* Here we use OIDs to denote components. OIDs consist of a sequence of identifiers that act as a path in a tree-structured name space. The identifiers can be either character strings or integers. A direct encoding of an OID in a message would be rather large. Therefore, we assume that components are grouped into logical sets such that their OIDs share a common prefix (e.g., see modifications to ASN.1 below). The OIDs of the types in a set would differ only in their last element, which would be an integer. This allows the OIDs in the assignments of a context list to be compressed by assigning a (small) integer identifier to each prefix and appending to the context list a table that maps each integer identifier to its prefix.

6. *Assign identifiers to representations for components.* Again, we use OIDs. The OID for a representation is obtained by appending a (small) integer identifier to the OID of the component that it represents. To encode a representation in an assignment that appears in a protocol message, we only need to encode its integer identifier.

7. *Changes to protocol.* There are three parts of the protocol that must be considered: connection establishment, default contexts, and altering the defined

context set (DCS). Both connection establishment and default contexts make use of OIDs that have been assigned to specific contexts, which act like named transfer syntaxes in these roles. The modifications to the three parts are as follows:

(a) *Connection establishment.* This part of the protocol actually remains unchanged. The initiator sends a list of OIDs for the named abstract syntaxes used by the application layer. For each named abstract syntax it also sends a list of OIDs for the named transfer syntaxes it supports for that abstract syntax. The responder chooses one transfer syntax for each abstract syntax and returns the choice to the initiator. In this capacity the named transfer syntaxes act as canonical contexts.

There are two reasons for leaving the connection protocol unchanged. The first is that all the data required for connection initiation and completion must fit into one message whose size is limited. This precludes sending a (possibly large) context list as part of connection establishment. The second is that it is compatible with the existing protocol, making it possible to introduce the new protocol gradually.

(b) *Default Contexts.* Default contexts are used primarily for expedited messages, which are out-of-band messages in that they are not synchronized with the normal message stream. A default context is required for expedited messages because the introduction, destruction, and use of presentation context identifiers is tightly coupled to the message stream. As in connection establishment, this part of the protocol uses a named transfer syntax as a canonical context.

(c) *Alter DCS.* The initial DCS is established at connect time and consists of the named abstract and transfer syntaxes agreed upon during connection establishment. The protocol allows the current DCS to be altered via the introduction and destruction of new presentation contexts. This is the part of the protocol where negotiation with context lists is introduced. The alter DCS functions are optional, so applications that do not want to use context lists are not forced to do so. In addition, there is no

restriction on the size of protocol messages, allowing the transmission of arbitrarily large context lists.

7.2.1.2 Changes to ASN.1

The following additions and changes would make ASN.1 fit the above model.

1. *Add a new language construct for defining abstract types.* One possible syntax is given by adding the following set of productions to those given in the ASN.1 standard [ISO87a]:

Typeassignment	→	*typereference* ::= *AbstractType*
AbstractType	→	*Tag* ABSTRACT { *RepList* }
RepList	→	*TaggedType* | *TaggedType RepList*

   where *Typeassignment*, *typereference*, *Type*, *Tag*, and *TaggedType* are as defined in the standard, and ABSTRACT, '::=', '{' and '}' are terminal symbols.

   There are three important points to observe. The first is that an abstract type definition can only appear as a type assignment, i.e., it cannot appear within the definition of another type. The second is that an abstract type has a tag. We require that no two abstract types defined in the same module have the same tag. The third point is that the tags of the tagged types in the representation list must be distinct.

2. *Require that each module have an OID.* Type definitions in ASN.1 are grouped into modules. This requirement is needed to ensure that each abstract type can be assigned an OID as outlined below, while allowing modules to be defined and checked separately.

3. *Define the function "components" for ASN.1.* The definition is similar to that for Amalgam given in Section 3.1.1. An abstract type definition defines a new component. Type assignments for types other than abstract types are treated like equates. Tagged types are treated as if they had no tags, i.e., the presence of a tag has no effect on the computation of the type's components. Macros are treated like equate constructors.

4. *Assign OIDs to built-in ASN.1 components and their representations.*

Fitting ASN.1 into the general model above requires that we assign OIDs to every abstract type and its representations. The OID for an abstract type is constructed by appending the tag in the abstract type's definition to the OID of the module in which the type is defined. The OID for an abstract type's representation is constructed by appending the tag from the tagged type that defines the representation to the OID of the abstract type.

Note that the above changes do not introduce abstract constructors. The closest approximation to an abstract constructor in ASN.1 is the macro notation, which is very complex because it allows the programmer to specify a grammar for new syntactic constructions that define types and values. Modifying the macro notation for the purpose of introducing abstract constructors is beyond the scope of this thesis. For the purpose of negotiation, the above description of ASN.1 treats macros in the same manner as equate constructors in Amalgam.

7.2.2 Negotiating for ROS

As defined in [ISO88b, ISO88c], ROS consists of an ASN.1 module that defines both a set of types that correspond to RPC protocol messages and a set of macros that are used in other modules to specify an RPC interface. The purpose of the macros is to provide a formal notation for specifying operations and errors [Whi89]. The macros, when expanded, do not define protocol messages, which are considered to be defined separately. Even though they are defined separately, the protocol messages are considered to be part of the same named abstract syntax as the module that imports and uses the macros. Because a context list is defined for a single named abstract syntax, grouping the protocol messages with the module using the macros means that a single context list must specify preferences for both the "application-level" and the protocol message types. From the standpoint of modularity, it would be better if the preferences were specified separately.

There is actually a small change to ROS that would allow the specification of two context lists, one for the protocol messages and one for the application. It is best illustrated by examining the following example protocol message type definition,

which defines an operation invocation message:

```
ROIVapdu  ::= SEQUENCE{
                invokeID invokeIDType,
                linked-ID [0] IMPLICIT invokeIDType OPTIONAL,
                operation-value OPERATION,
                argument ANY DEFINED BY operation-value OPTIONAL}
```

where a SEQUENCE is similar to a **struct** type in Amalgam and the keyword OPTIONAL means that a value for the field is not always required.

The key field to look at is **argument**, which contains the argument supplied by the application. The argument has type ANY, and the phrase DEFINED BY means that the actual type of the argument in a given invocation message depends on the value of the **operation-value** field.

The change consists of introducing a new data type that is similar to the type EXTERNAL, and then using this new type as the type of the **argument** field. The reason is that an EXTERNAL type indicates a change of context. The value of an EXTERNAL consists essentially of a context identifier and a value of any type. Inserting a context identifier into the data stream allows the value contained in an EXTERNAL type to be encoded using a different context than that used for the containing value. The reason a new type is needed is that the EXTERNAL type is actually more complex than described here and intended for a different purpose. In the case of remote operations we would like to have a DEFINED BY clause for the new type.

Given this new type, the protocol messages and application data types could be defined in separately named abstract syntaxes, each having its own context list and separately negotiated contexts.

7.3 Negotiation in OSF/DCE

Negotiating user-defined vspaces in the RPC facility of OSF/DCE is straight-forward given the design for Mercury. The reason is that OSF/DCE is essentially a subset of Mercury in that remote procedures are not first class objects like ports. As a result, the binding model for OSF/DCE only allows the construction of central servers. In

addition, OSF/DCE's interface description language contains a language construct for defining interfaces that is similar to the extension made to Amalgam in Section 7.1.1. Therefore, negotiable sets in OSF/DCE are associated with interfaces, just like in Mercury. Finally, OSF/DCE provides a naming service that is similar to the Mercury registry. Extending the naming service so that it can generate context UIDs should not pose any significant problems. Even if it did pose problems, negotiation could use context descriptions instead of context UIDs.

Besides adding negotiation with context lists to the presentation-layer, the only other major change is to extend the interface description language NIDL with a construct for abstract vspaces. Unlike Mercury, OSF/DCE does not allow distinct interface definitions to share vspace definitions. The scope of an abstract vspace definition is only the interface in which it is defined. Because interfaces are assigned UIDs, abstract components and their representations can be assigned IDs that are relative to the interface that contains them.

Recall from Section 7.1.4 that one of the major problems with Mercury is that modules may provide or use only a subset of an interface. This is only a problem for clients in OSF/DCE because servers always provide the entire interface. The solution here is the same as that for Mercury.

There is one additional problem in OSF/DCE that has not yet been addressed in Mercury, namely, the evolution of interfaces. OSF/DCE allows an existing interface to be superseded by a new version, as long as the new version is "compatible" with the old. In this case, compatible means that the new version can only provide additional remote procedures and vspaces. It cannot change the definition of the existing remote procedures or vspaces.

To allow evolution, OSF/DCE assigns strictly increasing version numbers to subsequent versions of an interface. A client must specify a version number when asking the name server to locate an instance of a given interface. The name server is allowed to substitute an instance of a newer version. The problem here is that the negotiable set of the requested version may be a proper subset of that of the substituted version. The solution proposed in Section 7.1.4 for dealing with subsets cannot be applied here because the client was compiled with a different negotiable set than the server. Therefore, the context UIDs used by the client will differ from

those of the server. The simplest solution is to require the server to provide context lists for both old and new versions. The name server would only substitute an instance of a newer version if the server exporting the instance could negotiate for the old version. The server can determine which version to use when the client actually contacts the server. Because clients initiate contact with servers, there is no additional round-trip communication required for determining which version of the context list to use.

Chapter 8

Veneer Design for Mercury

The purpose of this chapter is to show that negotiation is practical in that it can be integrated into an RPC-based system like Mercury without introducing substantial overhead on each remote procedure invocation. The overhead of negotiating a context is not as important because it is amortized over a number of invocations. The per invocation overhead, on the other hand, is important because it occurs on every invocation. This chapter shows that negotiation is practical by outlining a design for Mercury and examining the overhead per invocation. The design presented here is for an extension of an existing Mercury implementation without negotiation. Implementing this extension is beyond the scope of this thesis. However, enough detail is given to obtain a rough estimate of the overhead per invocation.

The implementation of a Mercury module consists of three layers: the Mercury system, the veneer, and the application program.[1] The Mercury system is the lowest layer and provides a low-level RPC mechanism. It hides the details of the RPC protocol and low-level communications issues such as buffering and synchronization. With the introduction of negotiation, it also hides the details of the negotiation protocol. However, it is not concerned with the representation of program data. The veneer sits on top of the Mercury system and provides a high-level, programming-language-specific RPC mechanism in the form of the port abstraction. It is also responsible for translating between the internal representation of the application

[1]Recall from Chapter 7 that a module corresponds roughly to an OSI application entity and incorporates the functions of the application, presentation, and session layers. The Mercury layers described here do not correspond to the OSI layers.

program's data and the transmission representation. The application program is the upper-most layer and uses the port abstraction provided by the veneer. The veneer, then, is a thin layer that bridges the gap between the low-level RPC mechanism and the high-level port abstraction.

The veneer consists of a number of stubs, one for each client and server port. Traditionally, stubs are concerned primarily with translating between the internal and transmission representations for application data. With the introduction of multiple representations and negotiation, stubs take on the additional role of coordinating translation with negotiation. For transmission, the context used for encoding is the *current context*, which is the context in effect at the time of transmission. For reception, the context used for decoding must be the same as that used by the transmitter for encoding.

Coordinating translation with negotiation is not trivial because the effective context may change during the course of communication. For example, a client may start communicating with a server using a canonical context while negotiating in the background. When negotiation is finished, both the client and the server must use the new context. The cost of coordination is the primary source of per invocation overhead.

Stubs perform translation by invoking procedures referred to as *XDR routines*. In coordinating the XDR routines with the effective context, a stub interacts with two subsystems: the negotiation subsystem and the XDR subsystem. The stub interacts directly with the negotiation subsystem to obtain the current context and to control negotiation at the request of the application program. Whenever the stub transmits a message, it must check the current context and select the correct XDR routine. Likewise, whenever a stub receives a message, it must determine the message's context and select the correct XDR routine.

Selecting the correct XDR routine on message transmission and reception can be rather expensive. Therefore, the XDR routines are cached. The details of caching are hidden by the XDR subsystem, which substantially simplifies the stub's structure.

As mentioned earlier, the design extends an existing implementation of Mercury. One major data abstraction from Mercury that occurs repeatedly in the description is the Mercury transport subsystem (abbreviated as *Hg-transport*), which imple-

ments almost all of the functionality of low-level Mercury RPC. It is represented as a single abstraction but is really an extensive body of code with a wide interface. Only those portions of the transport subsystem interface that are relevant to the operation of the stubs and negotiation are discussed here.

The rest of this chapter is divided into four sections, one each for the negotiation, XDR, client, and server subsystems. Each section describes the individual modules that constitute the subsystem and how they work together to accomplish a given task. The negotiation and XDR subsystems are described first because they are used extensively by the client and server stubs. A brief summary is included at the end.

In each section the description is accompanied by a module dependency diagram that illustrates the relationships between modules and the flow of information and control. Module dependency diagrams are described in [LG86], and an example can be found in Figure 8-1. In a module dependency diagram a data type abstraction is indicated by a box with an extra line at the top, and a procedural abstraction by a plain box. In both cases the abstraction's name is given inside the box. Arrows show which abstractions are used to implement a given abstraction. A solid arrow indicates that the implementation of the source abstraction (i.e., at the arrow's tail) depends on the target abstraction (i.e., at the arrow's head). If the target abstraction is a procedure then it is called by the source. If the target is a data type then either the source invokes the target's operations or the target is part of the source's representation. A dashed arrow indicates a weak dependence, which means that the source obtains a reference to the target, but does not use the target (e.g., by invoking its operations in the case of a data abstraction).

8.1 Negotiation Subsystem

The overall structure of the negotiation subsystem is shown by the module dependency diagram in Figure 8-1. The main abstraction is the *negotiator* data type, which is responsible for handling all negotiation for a given interface with a single Mercury peer. There is one negotiator instance for each combination of interface and peer. An interface is represented by an instance of the *interface-descriptor*

data type, which contains references to the interface's unique identifier and context list(s). Interface-descriptor instances are created by the stub compiler.

The negotiator has two responsibilities. The first is negotiating with a peer, which involves exchanging context lists and agreeing on a context and context identifier (CID). The second is deciding which context to use for encoding application-level data when a procedure invocation or reply message is sent. The following describes how these two responsibilities are met, along with a brief discussion of some implementation issues.

To carry out its first responsibility, the negotiator uses the Mercury transport subsystem to communicate with its peer. A Mercury peer is represented by an instance of the *module* data type, which encapsulates all the resources needed for communicating with the peer. The negotiator only references the module data type and does not use any of its operations. To send a message to a peer, the negotiator passes the message to the transport subsystem, together with the module reference for the peer. The transport subsystem uses the information in the module to send the negotiation message to the peer.

Reception of negotiation messages is handled somewhat differently. The main problem is that a negotiation message is received by the transport subsystem, which must then locate the negotiator for that message. Locating negotiators is the responsibility of the *negotiator-table* data type. The negotiator-table provides a mapping from interface UIDs to interface descriptors, and from module-interface pairs to negotiators. It is also responsible for creating negotiators and maintaining the invariant that there is at most one negotiator for each pair of module and interface descriptor. Because all negotiation is with respect to an interface, each negotiation message contains the UID of the interface to which it applies. When a negotiation message is received, the transport subsystem extracts the interface UID from the message. It then locates the negotiator by calling the negotiator-table with the interface UID and the module of the peer that sent the message. Finally, it passes the received message to the negotiator.

One point to notice is that the Mercury transport subsystem is responsible for delivering invocation and reply messages for ports as well as negotiation messages. Therefore, the transport subsystem presents two interfaces, one for stubs and an-

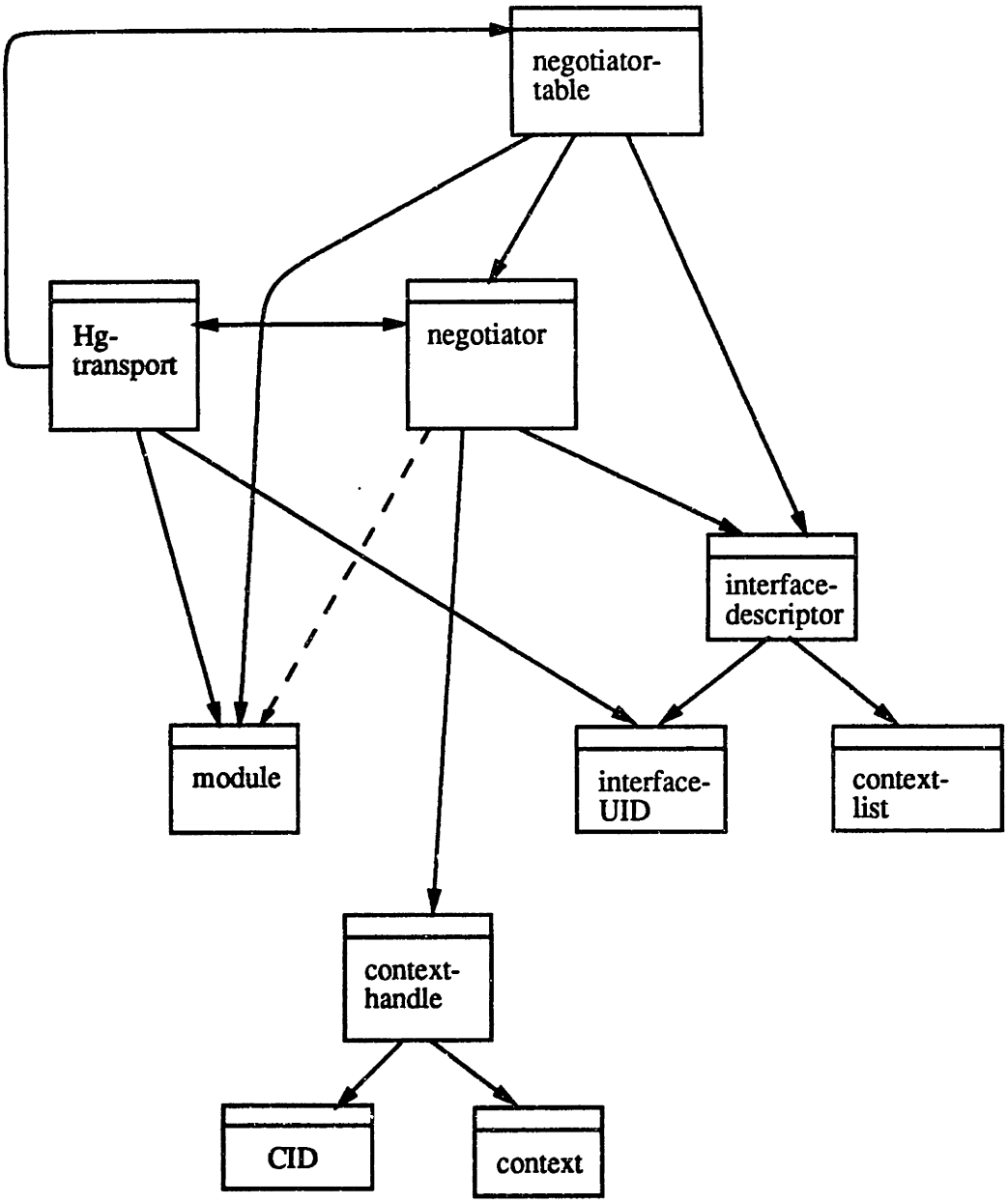


Figure 8-1: Negotiation subsystem module dependency diagram

other for negotiators. How it multiplexes negotiation and invocation/reply messages is an internal matter. This hides the structure of the underlying protocol(s) from the stubs and negotiator. The negotiator assumes that negotiation messages are delivered reliably and in order. The Mercury transport subsystem notifies the negotiator if it fails to deliver a message.

The second responsibility of the negotiation subsystem is deciding which context to use for encoding application-level data in procedure invocation and response messages. Negotiation is a process that occurs over time and, therefore, has state. The negotiator's view of the state is referred to as the *internal state*. A stub, on the other hand, needs to know the "current" context, that is, the context that should be used for encoding a message at any given point in time. Clearly, the current context is related to the current internal state. However, the internal state is at too low a level to be useful to a stub. Furthermore, making the internal state visible outside of the negotiator would greatly restrict future changes to its implementation. Therefore, an abstraction of the internal state, called the *external state*, is used to determine the current context. The external state is given by the following data type definition:

```
negotiation_state = oneof[canonical:null,
                          negotiating:null,
                          negotiated:null,
                          failed:null]
```

Each state has an associated context. The description of each state, including its associated context, is given below. A state transition diagram is shown in Figure 8-2. The following descriptions refer to operations of the negotiator data type, which is specified in Figure 8-3.

canonical - In this state the negotiator has no information about the peer's preferences or supported contexts. However, it can assume that the peer supports all canonical contexts for the interface. The associated context is the canonical context that is preferred by the negotiator's module. A transition to **negotiating** occurs via one of three events: the veneer (e.g., by request of the program) initiates negotiation by invoking the operation

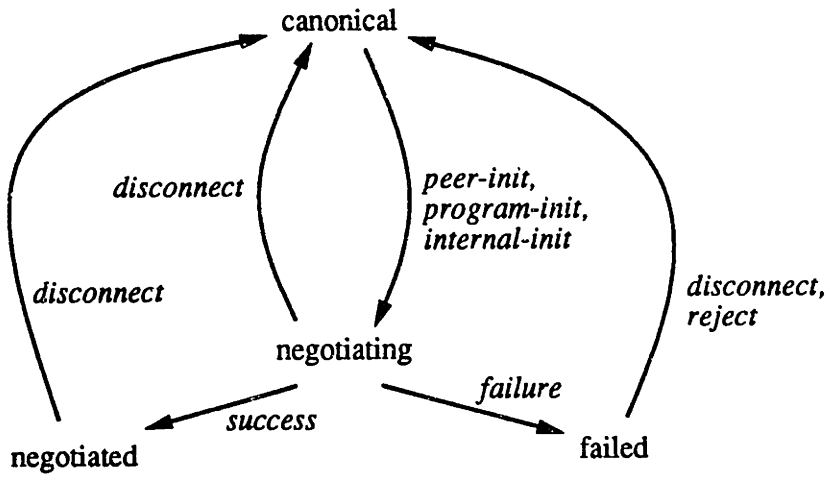


Figure 8-2: Negotiation state-transition diagram

`initiate-negotiation`, the negotiator receives a message from its peer to initiate negotiation, or the negotiator decides to initiate negotiation (e.g., by setting a threshold on the number of data messages that may be sent or received using the canonical context). These three events are referred to in the diagram as *program-init*, *peer-init*, and *internal-init*, respectively.

`negotiating` - In this state the negotiator is in the process of exchanging a context list with its peer and deciding on a context. The associated context is the preferred canonical context.

`negotiated` - This state is entered when negotiation completes successfully. The associated context is the context just negotiated. The negotiator remains in this state until a disconnect occurs, at which point it discards all information about its peer.

`failed` - This state is entered when negotiation fails to agree on a context other than the canonical context, which is also the associated context for this state. This state is distinct from `canonical` so that the program can observe the failure and avoid repeatedly negotiating and failing. The veneer (e.g. by program request) must explicitly reset the state to `canonical`.

The event *disconnect* in the state transition diagram exposes the design's only assumption about the negotiation protocol, namely that a negotiated context is as-

sociated with a time-span in which two peers share negotiation state. This time-span is referred to as a *negotiation connection.* When one module initiates negotiation, it establishes a negotiation connection with its peer. A negotiated CID is valid only for the duration of the connection in which it is negotiated. From the negotiator's point of view, a negotiation connection is not necessarily related to any underlying connection such as a session, transport, or network connection. For example, a negotiation connection could span multiple "sessions," thereby amortizing the cost of negotiation over a larger period of time. The details of connection management are hidden by the Mercury transport subsystem.

As part of its internal state, a negotiator keeps a context handle for the current context (see Figure 8-1). A *context handle* is a binding of a CID to a context. As discussed in detail in later sections, context-handles are used to coordinate the use of XDR routines. Stubs call the negotiator operation `get_current_context` to obtain the context handle for the context associated with the current state.

There are two important points with respect to the implementation of the negotiator and negotiator-table data types. The first is that the veneer is likely to be multi-threaded. In this case client and server threads run concurrently with threads initiated by up-calls from the low-level communications services of the Mercury transport subsystem. As a result, it is possible for the Mercury transport subsystem to call the negotiator or negotiator table concurrently with a stub. By design, the negotiator-table and negotiator are the only focal points of concurrency with respect to negotiation and, therefore, all synchronization is encapsulated in their implementation.

Second, because the negotiator provides the current context for use by the stubs and the current context can change, there must be a way of coordinating this change. The method chosen here is to have the stubs poll the negotiator every time they send an invocation or reply message (by calling the operation `get_current_context`). Another method is to use a call-back procedure whereby the stubs cache the current context and the negotiator calls the stub back when there is a change. The problem with this method is that it exposes the stub to potential synchronization problems because the stub and negotiator may be executing as different threads. Polling centralizes synchronization within the negotiator. The polling method is simpler

negotiator = **data type** is `create, get_state, get_current_context, initiate_negotiation, wait_on_negotiation, message_received, disconnect`

Operations

`create` = **proc**(interface-descriptor, module) **returns**(negotiator)

effects: Creates a new negotiator for the specified interface-descriptor and module. The initial state of the negotiator is `canonical`.

`get_state` = **proc**(negotiator) **returns**(negotiation_state)

effects: Returns the current state of the negotiator.

`get_current_context` = **proc**(negotiator) **returns**(context_handle)

effects: Returns the `context_handle` for the context associated with the current state.

`initiate_negotiation` = **proc**(negotiator) **signals**(bad_state)

modifies: negotiator

effects: If the current state is `canonical` then the state is set to `negotiating` and negotiation with the negotiator's peer is initiated. If the current state is `negotiating` or `negotiated`, then there is no effect. Signals bad_state if the current state is `failed`.

`wait_on_negotiation` = **proc**(negotiator) **signals**(not_negotiating, failed)

effects: Returns when the state is `negotiated`. Signals not_negotiating if the state is `canonical`. Signals failed if the current state is `failed` or if the on-going negotiation results in failure.

`message_received` = **proc**(negotiator, negotiation_message)

modifies: negotiator

effects: Notifies the negotiator that a negotiation message has been received from the negotiator's peer. Calling this operation implies the establishment of a negotiation connection.

`disconnect` = **proc**(negotiator)

modifies: negotiator

effects: Notifies the negotiator that the negotiation connection has been disconnected.

Figure 8-3: Specification of the negotiator data type

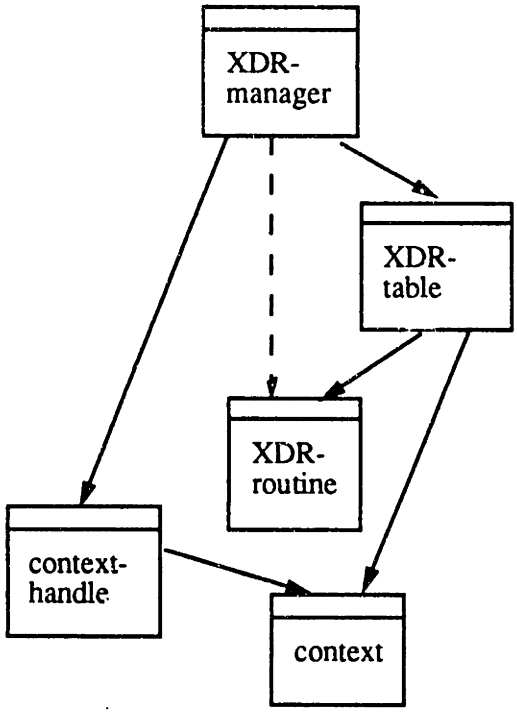


Figure 8-4: XDR subsystem module dependency diagram

and imposes very little overhead if the compiler can eliminate the procedure call via in-line substitution. The call-back method is more complex and should be used only if the procedure call for polling cannot be optimized.

8.2 XDR subsystem

The XDR subsystem is responsible for locating and caching XDR routines for use with a specific context. The structure of the XDR subsystem is shown by the module dependency diagram in Figure 8-4. Locating XDR routines is the province of the *XDR-table* abstraction and caching them is the province of the *XDR-manager*.

A given XDR routine marshals data values for a single data type. Because an argument list, normal result list, or exception result list is treated as a single data type, only one XDR routine is needed to marshal one of these lists. Therefore, for a given port and context there is exactly one XDR routine for each of the argument, normal result, and exception result lists.

When an interface is compiled, the stub compiler generates one XDR-table in-

stance for each argument, normal-result, and exception-result list of every port in the interface. Because a program may support many combinations of representations for the components of a vspace, an XDR-table may need to handle a large number of XDR routines. It is expected that some of the routines will be completely compiled procedures, while others will be table-driven or possibly interpreted in order to save space. The data structures for the table-driven and interpreted XDR routines could even be constructed at run-time. The main point here is that all these details are encapsulated within the XDR-table and are the province of the stub compiler and the XDR-table's implementation. To obtain an XDR routine for an argument, normal-result, or exception-result list of a port, one simply calls the lookup operation on the corresponding XDR-table instance, passing it the context under which the data is to be marshalled.

As discussed in other sections in this chapter, stubs look up XDR routines on every invocation and reply. Therefore, this look-up operation must be fast; anything more than a few instructions is too much. The XDR subsystem reduces the cost of looking up XDR routines by caching them.

The XDR-manager is the main access point for users of the XDR subsystem and is responsible for caching XDR routines. The cache is addressed by a context handle. That is, to obtain an XDR routine, one calls the XDR-manager and presents a context handle. Context handles are used instead of contexts because they have identity and, therefore, checking for a cache hit can be implemented by a simple address comparison. The negotiation subsystem and Mercury transport subsystem ensure that the same context handle is used to refer to a given binding of context to CID. In addition, each instance of a client or server stub has its own set of XDR-managers (see Sections 8.3 and 8.4). Because XDR-manager instances are not shared and there is only one current context per interface and peer module, an XDR-manager only needs a cache of size one. Checking for a cache hit requires very few instructions. In the case of a cache miss, the XDR-manager calls the lookup operation of the XDR-table.

8.3 Client Stub

From the application program's point of view, a Mercury port is represented by the port data type whose specification is given in Figure 8-5. There is one port data type for each port "x" that is defined in an interface. The remote procedure represented by the port is invoked using the `call` operation. This operation corresponds to the "client stub" found in traditional RPC systems. The `encode` operation is used when exporting a port, while `decode` is used when importing a port. The `create` operation is used by a server to make a procedure available as a remote procedure. It is discussed in Section 8.4.

As can be seen from the module dependency diagram in Figure 8-6, an instance of the port[2] data type encapsulates all the data needed by a client to invoke the remote procedure represented by the port. This data includes a reference to an instance of the type *Hg-port*, which is the low-level representation of the port used by the Mercury transport subsystem, the negotiator responsible for handling the port's contexts, and three XDR-managers, one each for the port's arguments, normal and exceptional results. The remainder of this section describes the control flow for invoking the port and focuses on how the XDR routines are coordinated with negotiation.

To invoke a port the application program invokes the `call` operation, passing it a port instance along with the arguments for the invocation (which are not shown in the figure). The operation first calls the negotiator to get the context handle of the current context for use with that port. It then calls the Mercury transport subsystem with the context handle, the Hg-port, the application program's argument list, a result vector that describes where to put the results, and the three XDR managers. The transport subsystem builds the message header, including the CID from the context handle. It also calls the XDR manager for the port's arguments with the context handle to get the XDR routine. It then calls the XDR routine to marshal the arguments into the message buffer and sends the invocation message.

When a reply is received, the Mercury transport subsystem first extracts the CID from the message and locates the corresponding context handle. Because the

[2]The prefix "x-" is omitted for brevity, but is understood as being present.

x-port = **data type is** `call, encode, decode, create`

Overview

An x-port represents the remote procedure named "x."

Operations

`call` = **proc**(x-port, *argument-list*) **returns**(*normal-result-list*)
signals(*exception-list*)

effects: Calls the remote procedure represented by the port. The form of the *argument-list*, *normal-result-list*, and *exception-list* depend on the definition of the remote procedure "x."

`encode` = **proc**(x-port, xdr-stream)

modifies: xdr-stream

effects: Encodes the port into the xdr-stream.

`decode` = **proc**(xdr-stream) **returns**(x-port)

modifies: xdr-stream

effects: Creates and returns a new port using the description encoded in the xdr-stream.

`create` = **proc**(**proctype**(*argument-list*) **returns**(*normal-result-list*)
signals(*exception-list*))
returns(x-port)

effects: Registers the specified procedure with the Mercury system as a remote procedure and creates and returns a new port representing that procedure. The form of the *argument-list*, *normal-result-list*, and *exception-list* of the specified procedure depends on the definition of "x."

Figure 8-5: Specification of the port data type

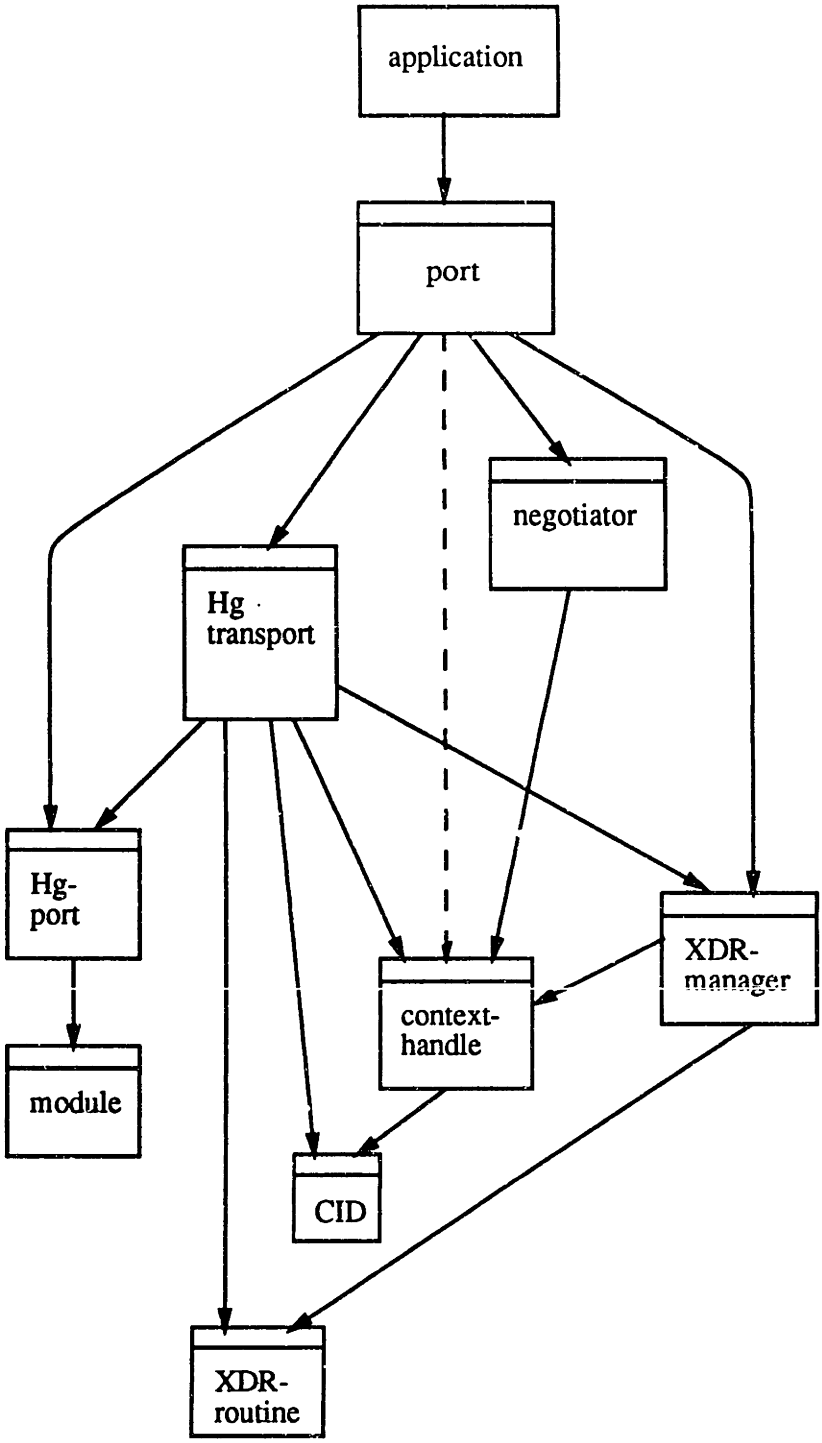


Figure 8-6: Module dependency diagram for client stub

CID of the reply message is usually the same as that of the invocation message, the transport subsystem already possesses the corresponding context handle, and locating it is trivial. In the case where the reply CID is different, the transport subsystem must look up the context handle in a special table that is not shown in the figure.

After it has located the context handle, the transport subsystem calls one of the XDR managers to obtain the corresponding XDR routine. Which XDR manager is called depends on whether the reply is for a normal or exceptional result. At this point the transport then calls the XDR routine to marshal the data from the message buffer (which is not shown in the figure) into the result vector that was passed by the `call` operation. Control is then returned to the `call` operation. The exact details of how control is returned depend on whether the call is a blocking call or stream call and are not discussed here. The `call` operation then returns the results to the application program.

Ports are created whenever they are imported from the "network." A port is imported either by retrieving an interface that contains that port from the Mercury catalog or by receiving the port as an argument or result of another port invocation. In all cases the port's description must be marshalled from an input buffer.

The `decode` operation is responsible for decoding port descriptions and, consequently, for creating an imported port. The implementation of the `decode` operation contains a static reference to both the interface-descriptor for the interface that contains the port and the XDR-tables for the port's arguments, normal results and exceptional results.[3] The operation first calls the decode operation of the Hg-port data type to create an Hg-port instance that is used to identify the port to the Mercury transport subsystem. It then obtains the module instance from the Hg-port and calls the negotiator table with the module and interface descriptor to locate the negotiator for the port. Finally, it creates an XDR-manager for each of the port's arguments, normal and exceptional results and then creates and initializes the port itself. The port is then returned to the application via whatever mechanism was invoked to import it.

[3]Recall from Sections 8.1 and 8.2 that the interface-descriptor and XDR-tables are generated by the stub compiler and, therefore, are known at compile time.

8.4 Server Stub

Although server stubs use many of the same data types as client stubs, the structure of a server is somewhat more complex. The reason is that servers must be registered before they can be invoked. For a client port, the peer is known at the time the port is imported, at which point all the required data structures can be initialized. Servers, on the other hand, do not know the identity of a client until the client's first invocation message arrives. Therefore, the negotiator and XDR-managers, which are specific to a given client, cannot be created until the first time the client invokes the port. Because of this, the creation of a server takes place in two stages, which are called registration and instantiation. The remainder of this section first describes how a remote invocation is processed. It then describes how a server stub is created and instantiated.

The specification of the server-stub data type is given in Figure 8-7. There is one server-stub data type for each port "x" that is defined in an interface. The operations `create` and `initialize` are discussed later. The operation `call` is used to process an invocation from a Mercury peer.

An invocation is processed in two steps. In the first step, the arguments are extracted from the received invocation message and the application procedure is called with those arguments. The context used to decode the arguments is determined by the CID in the received message. In the second step, the results returned by the application procedure are sent to the client in a reply message. The context used to encode the reply is the current context, which must be obtained from the negotiator.

The processing is illustrated by the module dependency diagram in Figure 8-8. When an invocation message arrives, the Mercury transport subsystem first extracts the port-id from the message and uses it to locate the server-stub in the port-table (not shown in the figure). The port-table maps a pair consisting of a port-id and module to a server-stub. Its initialization is discussed below. The transport subsystem then extracts the CID from the invocation message and locates the corresponding context-handle. Because the CID in an invocation is usually the same as for the previous invocation, the transport subsystem keeps the previous context-handle. This avoids having to locate the context-handle in most cases.

x-server-stub = **data type** is `create, initialize, call`

Overview

An x-server-stub represents a local procedure that can be called remotely from another Mercury module under the name "x."

Operations

`create` = **proc**(**proctype**(*argument-list*) **returns**(*normal-result-list*)
signals(*exception-list*))
returns(x-server-stub)

effects: Creates a new, initialized x-server-stub for the specified local procedure. The *argument-list*, *normal-result-list*, and *exception-list* of the specified procedure depend on the definition of "x."

`instantiate` = **proc**(x-server-stub, module) **returns**(server-stub)

effects: Returns a new x-server-stub that is initialized to accept remote calls from the specified Mercury module.

`call` = **proc**(x-server-stub, Hg-request-handle) **signals**(not-initialized)

modifies: x-server-stub

effects: Calls the local procedure with the arguments obtained from the Hg-request-handle, and sends a reply message with the results returned by the local procedure. Signals not-initialized if the `initialize` operation has not been called.

Figure 8-7: Specification of the server-stub data type

The transport subsystem then creates an Hg-request-handle to represent the call. The Hg-request-handle contains a reference to the context-handle together with any information needed to send a reply to the client. Finally, it invokes the `call` operation of the server-stub, passing it the Hg-request-handle.

The server-stub gets the XDR manager for the arguments from its internal state and sets up the argument vector (not shown in the figure) describing where the arguments are to be placed. It then calls the Hg-request-handle with the XDR manager to marshal the arguments. Finally, it calls the application procedure with those arguments, which completes the first step.

When the application procedure returns, the server-stub first gets the current context-handle from the negotiator. It then calls the Hg-request-handle for the reply, passing it the current context-handle, an XDR manager for the results (normal or exception) and a result vector (not shown in the figure) that describes where the results are located. The Hg-request-handle marshals the results into a message buffer and sends the reply message. This completes the second step.

As mentioned earlier, the creation of a server-stub instance occurs in two stages: registration and instantiation. When an application creates a port "x" for export, it calls the `create` operation of the x-port data type. The argument to the x-port `create` operation is the application procedure that the server-stub calls when a client invokes the port. The x-port `create` operation creates a new, uninitialized server-stub and registers it with the Mercury transport subsystem. When a server-stub is registered, the transport subsystem assigns it a port-id. An invocation message from a client contains this port-id as a reference. When the first invocation message with that port-id from a given client arrives, the transport subsystem instantiates a server-stub for that client and then enters it into the port-map. Subsequent calls from that client use the instantiated server-stub.

8.5 Summary

This chapter presents a design for extending an existing implementation of Mercury to provide negotiation. Implementing the extension mostly requires implementing the negotiation and XDR subsystems, modifying the Mercury transport subsystem

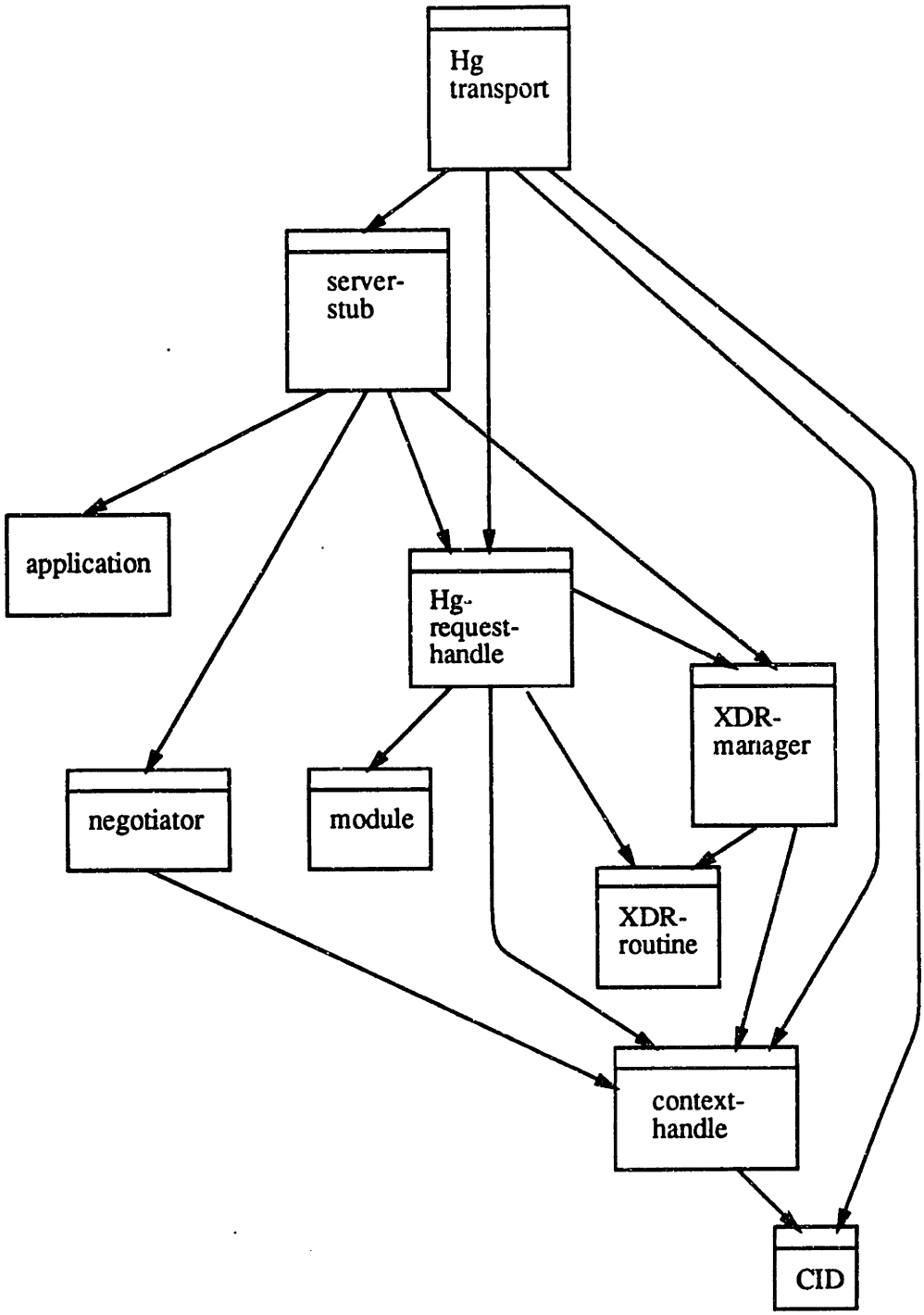


Figure 8-8: Module dependency diagram for server invocation

to multiplex negotiation messages with invocation/reply messages, and extending the stub compiler to generate interface descriptions, context lists, and XDR-tables. The changes to the stubs are actually quite small. If one removes the negotiator, XDR-manager, context-handle, and CID data types from Figures 8-6 and 8-8, then one arrives at the current Mercury design. The modifications to the existing stubs consist of adding a call to the negotiator to obtain the context-handle for the current context and changing the call to the transport subsystem so that it passes the XDR-manager instead of an XDR-routine. These modifications are rather simple.

Estimating the overhead per invocation of using negotiation requires examining the additional steps that are taken in the control path for transmitting and receiving an invocation or reply message. The additional steps for transmission are as follows:

1. The stub must call the negotiator to obtain the context-handle for the current context. I assume that the procedure call can be optimized via in-line substitution. The body of the procedure is relatively simple.

2. The transport subsystem must call the XDR-manager to obtain the XDR routine for the current context. Because XDR routines are cached, the overhead for this is small on average. Again, I assume that the procedure is substituted in-line.

3. The transport subsystem must encode the CID into the message buffer. For this I assume that the CID data type keeps a copy of the encoded CID. All that is required is to copy it into the buffer. I also assume that the encoded CID is small, on the order of a few bytes.

The additional steps for reception are as follows:

1. The transport subsystem must locate the context handle for the CID in the message. To make this fast on invocation, it saves the context-handle from the last invocation. For a reply, it saves the context-handle of the corresponding invocation. On average, the overhead consists of verifying that the CID of the saved context-handle matches that in the message. If the CID in the context-handle is in the encoded form, then the match is fast.

2. The transport subsystem must call the XDR-manager to obtain the XDR routine for the current context. The overhead here is the same as for sending an invocation/reply message.

In summary, the overhead for sending and receiving an invocation consists of five operations, each of which is quite simple in the average case. As a rough estimate, a total of fifty to one hundred "instructions" must be executed in addition to those required without negotiation. In terms of a RISC processor like the DECstation 5000, this translates into a few hundred CPU cycles. In comparison, the analysis in Chapter 6 estimates that a null RPC in a highly optimized system requires about 2000 CPU cycles. The additional overhead per invocation amounts to about 10-15% of the cost of an RPC without negotiation. To be fair, a null RPC does not include any overhead for marshalling arguments and results. In addition, the estimate of 2000 is quite generous. Therefore, the actual overhead of negotiation per invocation for a typical RPC is probably less than 10-15%.

Chapter 9

Conclusion

This dissertation has addressed the problem of specifying and negotiating efficient transfer representations for user-defined data types. User-defined data types can have a complex structure consisting of several layers of abstraction. The representation chosen at each layer can have a significant impact on the overall cost of marshalling a value. This chapter summarizes the new concepts introduced, analyzes the major design decisions, and gives suggestions for future research.

9.1 Summary

The core material of the dissertation is divided into three parts: the language for specifying and naming abstract representations for vspaces, a language for expressing preferences among contexts, and algorithms for choosing the best context given two sets of preferences. The remainder focuses on applying this material to real systems and showing that negotiating abstract representations is both useful and practical.

The most basic concept is the abstract vspace, which allows programmers to define new vspaces that have multiple abstract representations. An abstract vspace also has a specification that is independent of any representation. An abstract representation gives a structural definition of the vspace in terms of other (possibly abstract) vspaces. A transfer representation is obtained by specifying an abstract representation for each abstraction layer, together with a transfer representation for each built-in vspace and constructor. Each abstract representation for an abstract

vspace has a name that can be used for negotiation, putting user-defined vspaces on the same level as built-in vspaces. This is in direct contrast to all other interface definition languages, where only built-in vspaces have negotiable representations.

Presentation layer peers negotiate for the representation of a group of vspaces called a negotiable set. Negotiable sets are not directly expressible using language constructs in Amalgam. Instead, they are implicitly associated with higher-level constructs. The designs for both Mercury and OSF/DCE associate a negotiable set with a (remote service) interface, while the design for OSI uses a named abstract syntax. Associating negotiable sets with higher-level constructs relieves the programmer of having to designate each negotiable set specifically. It also simplifies the program interface presented to the application program by the presentation layer.

A context describes the transfer representation of every vspace in a negotiable set. In Chapter 3 we saw that it is often difficult to assign a quantitative cost measure to a context. In spite of this, programmers can specify qualitative preferences among contexts that reflect known cost differences. The biggest problem with specifying preferences is that there may be many contexts that interact with respect to cost. The proposed solution consists of a language that allows programmers to specify preferences for subsets of an interface and then combine the results. This is the well-known paradigm of "divide and conquer."

Negotiation for user-defined types can be made practical as shown by the algorithms in Chapter 6 and the design of the Mercury veneer in Chapter 8. In the common case where one peer supports the best context of the other, the cost of computing the best context is less than the cost of one RPC. Even in the worst case it is only on the order of several RPCs.

Finally, Chapter 6 points out that, given the current definition of a context, associating a negotiable set with an interface can result in inefficient negotiation in the case where a negotiable set contains many independent vspaces. One solution is to extend Amalgam with language constructs for expressing negotiable sets. This would allow the designer of an interface to divide the interface into several negotiable sets that are negotiated separately. The data presented in Chapter 6 suggest an alternative solution, which is to organize the structure of a context around vspaces

instead of vspace components, e.g., by specifying preferences for each vspace independently of any other vspace, and computing the best transfer representation for each vspace individually. This approach merits further research.

9.2 Analysis of Design Decisions

There are three primary decisions that were made in designing the mechanisms presented in this thesis. The first is to use a qualitative cost model based on preferences for negotiating the context with the lowest translation cost. The second is to remove the information from an encoded value that describes its representation (e.g., ASN.1 tags) and put it into a context. The third is to use the same representation for all occurrences of a given vspace component in the vspaces of a negotiable set. This section analyzes each of these decisions.

Using a Qualitative Cost Model

The reason for using a qualitative cost model is that it is much simpler than a quantitative model. Recall from Chapter 3 that quantitative models for marshalling user-defined vspaces are difficult to construct for several reasons: the presence of programmer supplied translation code, the uncertainty introduced by compiler optimization, the dependence on statistical properties of the data, and the need for accuracy. A complete model can be complex. For example, it must be capable of expressing and comparing the non-linear costs of translating VLSI mask descriptions. Given tools such as code analysis and runtime profiling programs, a quantitative model can be constructed if enough effort is expended.

In contrast, the qualitative model presented here is simple. As a general guideline, a programmer specifies that two contexts are ordered if one of them always has a lower cost than the other; they are specified as unordered otherwise. In spite of its simplicity, the model provides efficient communication in several important cases:

1. It avoids double translation in the case where two peers use the same internal representation.

2. It negotiates the context with lowest translation cost for a peer with higher

priority. This can be used to reduce the load on an important resource such as a server, or provide better response time for a resource-bound system that is connected to a resource-rich system, e.g., a personal computer connected to a mainframe.

3. It provides a simple form of load balancing when incorporated into mechanisms that dynamically adjust priority.

Its fundamental weak point, when compared to a quantitative model, is that it cannot express a cost measure that takes into account the translation cost of more than one peer. For example, it is not possible to negotiate the context with the lowest combined cost for two peers.

However, because of its simplicity, the model can be extended to handle an important case where the cost of communication includes the bandwidth of the communication medium. When bandwidth is low, it is desirable to compress data in order to save transmission time. In fact, this is one case where double translation may be required to achieve efficient communication. As with load balancing above, additional but simple mechanisms are needed. For example, an application would have two context lists: one for low bandwidth, and one for high bandwidth. The context list for low bandwidth would have the form $\mathcal{E}_c \gg \mathcal{E}_u$, where $\mathcal{E}_c$ expresses preferences among the compressed representations (e.g., the preference list for the video display image in Chapter 5), and $\mathcal{E}_u$ expresses preferences among the uncompressed representations. The context list for high bandwidth exchanges the two expressions. An application chooses one of the two context lists for negotiation based upon the available bandwidth. The crossover point between high and low bandwidth depends on the compression algorithm and the processor speed. A statically defined "lookup" table can be used to map processor speed to crossover point for a given compression algorithm.

A quantitative model that includes bandwidth has to either use the approach outlined above, or find a common currency (e.g., money) that relates bandwidth cost to translation cost. In the former case there is no advantage to using a quantitative model. In the latter case additional effort may be required to relate the two costs.

In summary, the qualitative model presented here handles many cases where

negotiation is needed to achieve efficient communication. It is easy to use and simpler than a quantitative model. Furthermore, computing the best context during negotiation is relatively fast. This is not to say that quantitative measures are less useful. There will always be cases where the extra effort of constructing a quantitative model is worth the gain. In addition, the quantitative measurements provided by analysis tools can be used to improve a qualitative model. Cases that were originally unknown or uncertain can be measured, and the results used to order contexts that were previously unordered. Ideally, instrumentation could be added to a program that would do this automatically. Other mechanisms that combine the best of qualitative and quantitative models are a subject for future research.

Binding Time of Representations

The second design decision is to remove descriptive information such as tags from the encoded values. Removing tags allows efficient block-move and direct-memory access (DMA) operations for marshalling data. In addition, marshalling procedures that perform translation do not need to examine a tag and perform different operations based on the tag's value. This results in more efficient marshalling procedures. In a sense, the representation of a value in a message is known "in advance," and the presentation compiler can take advantage of this to produce optimized code.

An analogous situation occurs in programming languages with respect to the time at which a name is bound to a value. For example, if one procedure calls another procedure that is known at compile time, then the compiler can perform interprocedural analysis to optimize the procedure call. On the other hand, if the called procedure is named by a procedure variable (which is bound at runtime), then the compiler cannot optimize the call. In the first case the procedure value is bound "early," while in the second case it is bound "late." The earlier a value is known, the more likely its use can be optimized. With respect to marshalling, early binding corresponds to putting the tags in the context, whereas late binding corresponds to putting them in the encoded values. By binding the representations of the values in a message prior to sending it, we can optimize marshalling for that message.

An even closer analogy is given by programming systems that use runtime code generation. As pointed out by Keppel, Eggers, and Henry [KEH91], there are a

number of cases where it is advantageous to compile code at runtime based on dynamic program values. In such cases, the dynamically optimized code can execute fast enough to pay for the runtime overhead of compiling it. Invoking the compiler at runtime is analogous to negotiating a representation. Just as the run-time overhead of compilation is compensated for by the gain in execution efficiency, the run-time overhead of negotiation is compensated for by the gain in marshalling efficiency. The analogy would be complete if the presentation compiler were invoked during negotiation to generate optimized code for the negotiated context. However, it is unknown at this point if runtime generation of marshalling code is feasible. The design presented in this thesis assumes that optimized code is generated statically for a number of commonly used contexts. The remaining contexts use interpreted code.

Returning to the issue of early versus late binding, late binding of representations within a message is not possible given the current design because there are no tags in the encoded values. In the case of programming languages there is an added advantage to providing late binding for procedures. The ability to pass procedures as parameters and results significantly enhances the expressive power of a programming language [ASS85, p. 67]. The corresponding question for marshalling is: are there any advantages to be gained by putting information like tags into the encoded values?[1]

There is at least one case where adding information results in more efficient translation. Consider the case illustrated in Figure 9-1, which represents a client using a distributed database. In the Figure, peer B is a front end for a database that is distributed over peers C and D. B accepts a query from client A, analyzes it, and distributes specialized queries to C and D, which then return the data to B. B merges the data from C and D and passes the result to A. Assume that the result is represented by the vspace

bounded_sequence[n,**struct**[...]]

where some of the elements of the sequence come from C and some from D. As-

[1]Others have argued that the redundancy provided by tags permits the receiver to verify that the values have the correct type and are encoded correctly. This is a separate issue from that discussed here.

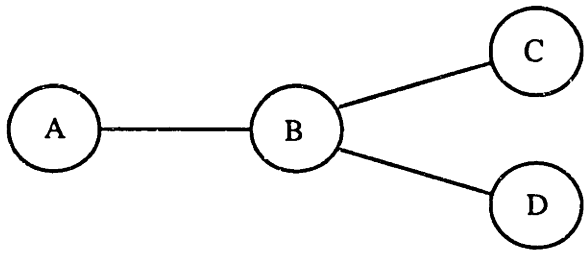


Figure 9-1: Example of a distributed database

sume also that the merge arbitrarily orders the elements from C and D within the sequence. Now, it is possible that C and D run on different machines and use different representations for some component of some field within the **struct**. If B, C, and D are to avoid translating data, then the message to A must indicate the encoding used for each element in the sequence. This requires adding information to the encoded value of each element. Possible solutions that allow late binding where needed are discussed in the next section on future research.

Multiple Occurrences of Vspace Components

The final design decision is to use the same representation for every occurrence of a given vspace component in the vspaces of a negotiable set. Again, this decision was made for reasons of simplicity. The following vspace definition illustrates the primary limitation imposed by this decision:

struct[one: rectangle, two: rectangle]

It is possible that a program uses a different internal representation for each of the two rectangles. In this case, the context with the lowest translation cost cannot be described. One can argue that this example is unlikely because programs usually use one internal representation for all values of a given type.

Let us assume for the moment that there are more compelling examples for which this limitation is unacceptable, and look at possible solutions. One solution is to make contexts more expressive, for example, by allowing them to refer to specific instances of components within vspace expressions. However, this increases the complexity of assignments as well as the number of possible contexts for a given

negotiable set. The result is to increase the cost of negotiation and the complexity of the language for describing context lists.

Another alternative is to make negotiable sets arbitrarily small and insert context identifiers into the data stream as opposed to having one at the beginning. This would also solve the problem presented by the distributed database example above. However, we now must negotiate for several negotiable sets. When taken to its highest level of flexibility, every component will have its own context identifier. This has the effect of re-introducing tagging as found in the basic encoding rules for ASN.1. Clearly there is some point where the added cost of inserting CIDs is worthwhile, but it is not clear *a priori* just where that point lies.

In short, no matter what design choice one makes, there will be cases where the cost of negotiation out-weighs that of communication or vice versa. In essence, there is a fundamental tradeoff between efficient negotiation and efficient communication, and no fixed choice is best in all cases. Only the application knows when it is worth spending more effort on negotiation in order to achieve more efficient communication. Allowing the programmer or the application to control negotiation to this extent requires adding functionality to the presentation layer's program interface and the languages for defining vspaces and preferences. Whether or not it is possible to add this functionality without significantly increasing the complexity of using it is unknown.

9.3 Suggestions for Future Work

There are several possible extensions to the work presented in this thesis. The first is suggested by the analysis at the end of the previous section. What is needed are mechanisms that give the programmer control over the tradeoff between the efficiency of negotiation and of communication. As a first step one could add language constructs for expressing negotiable sets.

Another language construct would express that the context for a portion of a vspace should be negotiated separately. For example, we could introduce a constructor called **negotiable** with the following structure:

negotiable[*handle*, *Vspace-exp*]

where *handle* is an identifier that provides a name that can be referenced by negotiation. The meaning of an instance of this constructor consists of the vspace given by *Vspace-exp*. In addition, it defines a negotiable set consisting of just that vspace. As an example of its use we could have

struct[one: **negotiable**[foo,rectangle],
two: **negotiable**[bar,rectangle]]

which indicates that each component can be negotiated separately.

Another possible language construct would allow the programmer to designate points where a context identifier is to be inserted into the data stream. For example, consider the constructor

dynamic[*n-set*, *Vspace-exp*]

where *n-set* is an identifier that references a named negotiable set and *Vspace-exp* is a vspace in that set. The context for a **dynamic** vspace is separate from that of the vspace that encloses it. The encoding of such a vspace includes a context identifier. For example, in the database example in the previous section we would use the following vspace:

bounded_sequence[*n*,**dynamic**[baz,**struct**[...]]]

in which case every element of the sequence would carry its own CID.

As mentioned in Chapter 7, Amalgam cannot be used to describe the protocol headers of Mercury invocation and response messages. One of the reasons is that a protocol header contains a CID, which cannot be expressed in Amalgam. A constructor similar to **dynamic** could provide a potential solution to this problem, although there are still issues to be addressed, e.g., the linkage between the specific vspace and other values in the header such as the message kind (e.g., invocation/response) and the port identifier. This is similar to the approach suggested for OSI in Section 7.2.

Another area for research is to make Amalgam an extensible language that allows the definition of complex constructs like ports and interfaces. A simple macro facility like that found in ASN.1 would not be sufficient because it must interact with the

definition of the function *components* (e.g., see Section 7.1.2). The challenge is to create an easily used language that allows programmers not only to design and use application-level protocols, but also to control the tradeoff between the cost of negotiation and the cost of communication.

One important area of research is developing quantitative cost models for marshalling a data structure given a specific programming language, compiler and processor. A quantitative model would allow a wider range of goals for negotiation. For example, we could negotiate the context that has the lowest total cost for both peers, the lowest response time, or the lowest bandwidth requirements.

Although the semantics of converting a value between representations have not been considered in this thesis, there are a number of important issues that must be addressed. For example, conversion between different floating point formats does not always preserve the exact value. Similarly, converting a point value between cartesian coordinates and polar coordinates may not be value preserving. This topic is discussed briefly in Appendix B.

Finally, there is one very important related area that has not been mentioned at all in the thesis, namely, the specification of marshalling procedures that are both modular and efficient. For example, in the case of matrix_2d we would like to write a single parameterized encode/decode procedure that handles matrices of any element type. When given an encode/decode procedure for the element type, the system would produce code that is as optimal as the hand-generated code in appendix A.

The approach taken by Argus and detailed by Herlihy and Liskov [HL82] is very attractive because it allows programmers to write modular encode/decode procedures for each abstract data type, which are then linked together. It also abstracts away the low level details of managing buffers and allows the programmer to express directly the transformation of the data structure. However, Liskov's and Herlihy's design assumes that each abstract vspace has exactly one representation. In addition, it also creates additional storage to hold a transformed data structure before actually putting it into a buffer. Extending the design to overcome these limitations is certainly worth further research.

Appendix A

Matrix Marshalling Code

This appendix contains the procedures that were used to make the measurements given in Section 1.1. The code and definitions presented here are based on an existing, low-level message interface for an implementation of Mercury. The code consists primarily of the four procedures written in C that are shown in figures A-2 through A-5. The main procedure, which is not shown, performs the following: 1) parses command-line arguments specifying the matrix size, buffer size and procedure to use, 2) initializes the data structures, and 3) invokes the specified procedure. Common definitions used by all the procedures are given in figure A-1.

The primary interface to the underlying buffer mechanism is via the data structure **message** shown in figure A-1. Besides containing a reference to the current buffer and a window into that buffer, it contains a reference (**getnext**) to a procedure that is called to obtain a new buffer. This procedure is called by the marshalling procedures after a buffer is filled. The procedure used here simply resets the window into the current buffer. Therefore, the measurements include the procedure-call overhead of breaking the matrix into message packets, but not the overhead of allocating and deallocating buffers.

The structure of each procedure is the same: an outer loop that obtains message buffers and an inner loop that fills each buffer. The main difference between procedures is in the inner loop. In the case of block move (figure A-2), the inner loop is performed by the procedure **bcopy**, which is part of the standard C environment. All procedures assume that the initial window into the buffer starts on an **int32**

boundary (i.e. no alignment neccessary), and that the buffer's size is an integral number of **int32** values.

Two hand optimizations have been applied to eliminate variation due to the ability of different compilers to optimize code. The first is that the procedure `swapbytes` is expanded in line instead of called. The second uses strength reduction to eliminate multiplication in the inner-loop of the column-major procedures.

The testbed for measurements is the same as that given in Section 6.7.1. The procedures were compiled with GCC version 2.1 and run on a Decstation 5000/200. The utility Pixie was used to count the number of CPU cycles consumed by each procedure. The number of cycles required to marshal a matrix element consists of the total number consumed by the procedure divided by the number of elements in the matrix. Measurements were taken for buffer sizes of 512 and 1024 bytes and for matrix sizes ranging between 10x10 and 1000x1000 elements. The figures presented in Section 1.1 are an average computed from these measurements.

```
/* the following definitions are for simulating a message system */

typedef int _Hg_INT32;
typedef char Byte;

struct Window {
  Byte *current, /* ptr to next location in buffer */
  Byte *maxp;    /* ptr to end of window in buffer
};

typedef struct buffer {
  Byte *data;    /* ptr to allocated buffer */
  int length;    /* length of allocated buffer */
  struct Window window;  /* window into allocated buffer */
} *Buffer;

typedef struct message {
  /* procedure to get next buffer */
  struct message * (*getnext)(struct message *, int);
  struct Window *window;  /* window into current buffer */
  Buffer current_buffer;
  int buf_size;   /* number of bytes to allocate */
} *Message;

/* "inline" procedure to swap bytes of int32 */
#define swapbytes(n) ((n << 24) & 0xff000000) | \
                     ((n << 8)  & 0x00ff0000) | \
                     ((n >> 8)  & 0x0000ff00) | \
                     ((unsigned) n >> 24)
```

Figure A-1: Common definitions for procedures

```
void put_matrix_int32(Message msg, _Hg_INT32 *matrix, int num_rows,
                      int num_cols)
{
  int num_bytes, buf_length, bytes_to_move;
  char *byte_array;
  byte_array = (char *) matrix;
  /* compute number of bytes in matrix */
  num_bytes = num_rows * num_cols * sizeof(_Hg_INT32);
  while (num_bytes > 0) {
    if (msg->window->current >= msg->window->maxp)
      msg =  (*(msg->getnext))(msg,sizeof(_Hg_INT32));
    /* block copy - move as many bytes as will fit into buffer.*/
    buf_length =  msg->window->maxp - msg->window->current;
    bytes_to_move =
          (buf_length < num_bytes) ?  buf_length : num_bytes;
    bcopy(byte_array, msg->window->current, bytes_to_move);
    byte_array += bytes_to_move;
    msg->window->current += bytes_to_move;
    num_bytes -= bytes_to_move;
  }
}
```

Figure A-2: Procedure for row-major without byte swapping

```
void put_matrix_int32_swap(Message msg, _Hg_INT32 *matrix,
                           int num_rows, int num_cols)
{
  int buf_length;
  int elems_to_move, num_elems;
  _Hg_INT32 *bufp;

  num_elems = num_rows * num_cols;
  while (num_elems > 0) {
    if (msg->window->current >= msg->window->maxp)
      msg =  (*(msg->getnext))(msg,sizeof(_Hg_INT32));

    buf_length = (msg->window->maxp - msg->window->current)
                 /sizeof(_Hg_INT32);
    elems_to_move =
         (buf_length < num_elems) ?  buf_length : num_elems;
    bufp = (_Hg_INT32 *) msg->window->current;
    num_elems -= elems_to_move;

    while (elems_to_move--)
      {
        *(bufp++) = swapbytes(*matrix);
        matrix += 1;
      }
    msg->window->current = (Byte *) bufp;
  }
}
```

Figure A-3: Procedure for row-major with byte swapping

```
void put_cm_matrix_int32(Message msg, _Hg_INT32 *matrix,
                         int num_rows, int num_cols)
{
  int buf_length;
  int elems_to_move, num_elems;
  int col_count = 0;
  _Hg_INT32 *matrix_save = matrix;
  _Hg_INT32 *wrap_check;
  _Hg_INT32 *bufp;

  num_elems = num_rows * num_cols;
  wrap_check = matrix + num_elems; /* point past matrix */
  while (num_elems > 0) {
    if (msg->window->current >= msg->window->maxp)
      msg =  (*(msg->getnext))(msg,sizeof(_Hg_INT32));

    buf_length =  (msg->window->maxp - msg->window->current)
                  /sizeof(_Hg_INT32);
    elems_to_move =
        (buf_length < num_elems) ?  buf_length : num_elems;
    bufp = (_Hg_INT32 *) msg->window->current;
    num_elems -= elems_to_move;

    while (elems_to_move--)
      {
       *(bufp++) = *matrix;
       matrix += num_cols;    /* advance to next row */
       if (matrix >= wrap_check) matrix = matrix_save +  ++col_count;
      }
    msg->window->current = (Byte *) bufp;
  }
}
```

Figure A-4: Procedure for column-major without byte swapping

```
void put_cm_matrix_int32_swap(Message msg, _Hg_INT32 *matrix,
                              int num_rows, int num_cols)
{
  int buf_length;
  int elems_to_move, num_elems;
  int col_count = 0;
  _Hg_INT32 *matrix_save = matrix;
  _Hg_INT32 *wrap_check;
  _Hg_INT32 *bufp;

  num_elems = num_rows * num_cols;
  wrap_check = matrix + num_elems; /* point past matrix */

  while (num_elems > 0) {
    if (msg->window->current >= msg->window->maxp)
      msg =  (*(msg->getnext))(msg,sizeof(_Hg_INT32));

    buf_length =  (msg->window->maxp - msg->window->current)
                  /sizeof(_Hg_INT32);
    elems_to_move =
        (buf_length < num_elems) ?  buf_length : num_elems;
    bufp = (_Hg_INT32 *) msg->window->current;
    num_elems -= elems_to_move;

    while (elems_to_move--)
      {
       *(bufp++) = swapbytes(*matrix);
       matrix += num_cols;    /* advance to next row */
       if (matrix >= wrap_check) matrix = matrix_save +  ++col_count;
      }
    msg->window->current = (Byte *) bufp;
  }
}
```

Figure A-5: Procedure for column-major with byte swapping

Appendix B

Abstract Vspace Conversion Issues

The video display image abstraction in Chapter 5 raises interesting questions about the relationship between different representations for the same abstract vspace. Previously, I had only considered multiple representations for which there is a one-to-one mapping between the values of every pair of representations, where corresponding values denoted *exactly* the same abstract value. An example of an abstract vspace with multiple representations that does not fall into this category is the vspace "point" with cartesian and polar coordinate representations. The problem with cartesian and polar coordinates is that they measure a continuous medium with finite precision and the conversion between them involves trigonometric functions. As a result, there are values in one coordinate system that do not correspond exactly to values in the other, although a one-to-one mapping might be constructed.

A similar situation exists with the video display image in that the RGB and YUV coordinate systems do not represent the exact same set of color values. Conversion between them involves multiplication and division by fractional values and rounding. However, the abstraction is based on a visual image that is displayed on a physical device and perceived by a person, which is somewhat qualitative. Even if the mapping were exact, an image does not look exactly the same when displayed on different devices without careful calibration and (possibly) sophisticated compensation. In other words, an abstract value of the pixel vspace denotes an "interval" in

an abstract color space and not an exact point. In spite of this, the abstraction is still useful as evidenced by the use of RGB–YUV conversion in television transmission systems and the willingness of people to use image conversions that are far less faithful than those between RGB and YUV.

The difference between using abstract vspaces with multiple representations and having users invoke specific conversions is that, in the former case, the system is hiding the conversion and claiming that the abstraction is being faithfully maintained. In order to be faithful, the abstraction's designer must be careful to make the abstraction's specification consistent with the conversion between representations. In particular, the designer must take into consideration the effects of repeated conversion. There are essentially four categories of abstraction with respect to repeated conversion:

1. *Perfect conversion.* An example is the matrix_2d[**int32**] vspace from Chapter 2. In this case there is no effect due to repeated conversion, and the designer can define the abstraction in terms of exact abstract values.

2. *Convergence.* An example is the video display image where the representations are restricted to RGB and YUV with the same resolution and there is no compression. In experiments that are not reported here, I have observed that repeated conversion converges to a single value within less than ten iterations, where the average difference between the initial and final values is qualitatively small (i.e., on the order of less than 1% on average). It is unknown whether DCT compression without quantization converges with repeated cycles of compression and decompression. If it does, then it also falls in this category.

   The uncompressed RGB, YUV, YUV:4:1:1, and YUV:4:2:2 representations for the video display image can also be put in this category if the filter operation is the exact inverse of the interpolation operation.[1] The reason this works is that information is lost only on the first conversion from RGB to YUV:4:n:n or

[1] One cannot make interpolation be the exact inverse of filtering because filtering can remove information. However, it is possible to make filtering be the exact inverse of interpolation in this case.

from YUV to YUV:4:n:n. Subsequent conversions do not lose any additional information.

In this category, the abstract values are "intervals." Whether or not the intervals can be defined in terms of precise abstract values may depend on the given abstraction. In the video display image, the intervals correspond to sets of images whose pixels fall into equivalence classes, where an equivalence class consists of all RGB and YUV values that converge to the same value.

3. *Divergence.* In this case information is lost with each conversion, and the original abstract value deteriorates to the point that it is useless. An example is a compression method that loses information with each compression-decompression cycle. Such abstractions may still be useful in spite of the loss. For example, we could define a transmitable video image abstraction that can be used to transmit an image from a source, such as a camera or image archive, to a display device for viewing, where the number of conversion cycles is small. This is essentially what happens with television transmission systems and video cassette recorders.

4. *Hybrid.* This is essentially a catch-all category for conversions that exhibit a combination of properties. It is not clear if such abstractions are useful at this point.

In summary, the specification of an abstract vspace must take into account the implementation of conversion between its representations. In addition, it must make clear the circumstances under which the abstraction may be used.

Bibliography

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill, 1985.

[BCL+87] Brian N. Bershad, Denis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[BT80] Clark M. Baker and Chris Terman. Tools for verifying integrated circuit designs. *Lambda*, pages 22–30, 1980. Fourth Quarter.

[CM88] Fausto Caneschi and Emanuela Merelli. An architecture for an ASN.1 encoder/decoder. *Computer Networks and ISDN Systems*, 11:297–303, 1988.

[Coh81] Danny Cohen. On holy wars and a plea for peace. *IEEE Computer*, 14(10):48–54, October 1981.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Communications Architectures & Protocols; SIGCOMM '90 Conference*, pages 200–208. ACM, 1990.

[DECa] Digital Equipment Corporation. *pixie(1).* Ultrix 4.2 General Information, Vol. 3B (Commands(1): M-Z).

[DECb] Digital Equipment Corporation. *prof(1).* Ultrix 4.2 General Information, Vol. 3B (Commands(1): M-Z).

[DLM+87] Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant. The Network Computing Architecture and System: An environment for developing distributed applications. In *Summer USENIX Technical Conference, Phoenix, AZ.*, pages 385–398. USENIX Association, June 1987.

[FS91] Dietmar Fauth and Chikong Shue. Remote procedure call: Technology, standardization and OSF's distributed computing environment. In *Proceedings of ONLINE '91, Hamburg, Germany*, 1991. Also published as a technical paper by the Open Software Foundation.

[GHW85a] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch family of specification languages. *IEEE Software*, 2(4), September 1985.

[GHW85b] John V. Guttag, James J. Horning, and Jeannette M. Wing. Larch in five easy pieces. SRC Report 5, Digital Equipment Corp., Systems Research Center, July 1985.

[Gib87] Phillip B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, 13(1):77–87, January 1987.

[HA90] Saleem M. Haider and Peng H. Ang. A data-structuring technique for gridded VLSI layouts. In *Proc. of the European Design Automation Conference*, pages 356–362. IEEE Computer Society Press, March 1990.

[HD89] Christian Huitema and Assem Doghri. A high speed approach for the OSI presentation protocol. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks.* Elsevier Science Publishers B.V. (North-Holland), 1989.

[HL82] Maurice Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[HS87] R. Hayes and R. D Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, 13(12):1254–1264, December 1987.

[ISO84] International Organization for Standardization. *ISO 7498 - Open Systems Interconnection - Basic Reference Model*, 1984.

[ISO87a] International Organization for Standardization. *ISO 8824 - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)*, 1987.

[ISO87b] International Organization for Standardization. *ISO 8825 - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, 1987.

[ISO88a] International Organization for Standardization. *Information Processing Systems - Open Systems Interconnection - The Directory. DIS 9594*, April 1988.

[ISO88b] International Organization for Standardization. *ISO DIS 9072-1.2 - Remote Operations - Part 1: Model, notation and service definition*, 1988.

[ISO88c] International Organization for Standardization. *ISO DIS 9072-2.2 - Remote Operations - Part 2: Protocol Specification*, 1988.

[ISO88d] International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). *Final Text of DIS 8822, Information Processing Systems - Open Systems Interconnection - Connection Oriented Presentation Service Definition*, April 1988.

[ISO88e] International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). *Final Text of DIS 8823, Information Processing Systems - Open Systems Interconnection - Connection Oriented Presentation Protocol Specification*, April 1988.

[ISO91] International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). *ISO/IEC JTC1 Committee Draft 10918-1, Digital Compression and Coding of Continuous-Tone Still Images, Part1, Requirements and Guidelines*, February 1991.

[Jam90] David V. James. Multiplexed buses: The endian wars continue. *IEEE Micro*, 10(3):9–21, June 1990.

[Jos88] Anthony D. Joseph. Run-time negotiation of value representation in a distributed system. Master's thesis, Massachusetts Institute of Technology, June 1988.

[JR86] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In *OOPSLA '86 Conference Proceedings*. ACM, September 1986.

[JRT85] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: An interface specification language for distributed processing. In *Proc. 12th ACM Symp. on Principles of Program. Lang.*, January 1985.

[Ked82] Gershon Kedem. The quad-CIF tree: A data structure for hierarchical on-line algorithms. In *Proc. of the 19th ACM/IEEE Design Automation Conference*, pages 352–357, June 1982.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 1991.

[LBG+88] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the Mercury system. In Bruce Shriver, editor, *Proc. 21st Annual Hawaii International Conference on System Sciences*, volume 2, January 1988.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986. ISBN 0-262-12112-3.

[Lis87a] Barbara Liskov. Modules, agents, sequence identifiers, and ports. Mercury Design Note 17, Programming Methodology Group, Laboratory for Computer Science, MIT, Cambridge, MA 02139, January 1987.

[Lis87b] Barbara Liskov, et. al. Argus reference manual. Technical Report MIT/LCS/TR-400, MIT, 1987.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[LW86] Barbara Liskov and William Weihl. Data types and communication. Mercury Design Note 2, Programming Methodology Group, Laboratory for Computer Science, MIT, Cambridge, MA 02139, May 1986.

[MC80] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[Mis85] Frederick C. Mish, editor. *Webster's Ninth New Collegiate Dictionary.* Merriam-Webster Inc., 1985.

[Mur90] Erin E. Murphy. Reconciling conflicting design-automation standards. *IEEE Spectrum*, pages 44–45, March 1990.

[NBL+88] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):258–272, March 1988.

[NH88] Arun N. Netravali and Barry G. Haskell. *Digital Pictures: Representation and Compression.* Plenum Press, 1988.

[NR86] S. K. Nandy and I. V. Ramakrishnan. Dual quadtree representatin for VLSI designs. In *Proc. of the 23rd ACM/IEEE Design Automation Conference*, pages 663–666, June 1986.

[Ous84] John K. Ousterhout. Corner stitching: A data structuring technique for VLSI layout tools. *IEEE Transactions on CAD/ICAS*, CAD-3(1):87–99, January 1984.

[Ous85] John K. Ousterhout, et. al. The Magic VLSI layout system. *IEEE Design & Test*, pages 19–30, February 1985.

[Sam88] Hanan Samet. Hierarchical representations of collections of small rectangles. *ACM Computing Surveys*, 20(4):271–309, December 1988.

[SB89] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. In *Proc. 12th ACM Symposium on Operating Systems Principles*, 1989.

[Sun85] Sun Microsystems, Inc. *Remote Procedure Call Specification*, 1985.

[Sun87] Sun Microsystems, Inc. *XDR: External Data Representation Standard*, June 1987. Also available as Internet RFC 1014, Network Information Center, SRI International.

[vRvST89] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. The performance of the Amoeba distributed operating system. *Software Practice & Experience*, 19(3):25–34, October 1989.

[Wal91] Gregory K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4), April 1991.

[Wei87a] William Weihl. Communicating values with sharing structure. Mercury Design Note 14, Programming Methodology Group, Laboratory for Computer Science, MIT, Cambridge, MA 02139, January 1987.

[Wei87b] William Weihl. Polymorphism. Mercury Design Note 15, Programming Methodology Group, Laboratory for Computer Science, MIT, Cambridge, MA 02139, January 1987.

[Wei88] William Weihl. Base vspaces and the interface description language. Mercury Design Note 29, Programming Methodology Group, Laboratory for Computer Science, MIT, Cambridge, MA 02139, September 1988.

[Whi89] James E. White. ASN.1 and ROS: The impact of X.400 on OSI. *IEEE Journal on Selected Areas in Communications*, 7(7):1060–1072, September 1989.

[WSG91] Yi-Hsiu Wei, Alexander D. Stoyenko, and German S. Goldszmidt. The design of a stub generator for heterogeneous RPC systems. *Journal of Parallel and Distributed Computing*, 11:188–197, 1991.

[Xer81] Xerox Corporation. Courier: The remote procedure call protocol. Technical Report XSIS 038112, Xerox Corporation, December 1981.